MASTER EN SCIENCES INFORMATIQUES

Implémentation d'un langage générique de programmation

Vansimpsen, Tom; Letocart, Vincent

Award date:
1998

Mémoire en vue de l'obtention du grade de
licencié en informatique

Implémentation d'un langage générique de programmation

Tom Vansimpsen & Vincent Letocart

Promoteurs : Naji Habra & Baudouin Le Charlier

Année académique 1997-1998

Résumé

Ce travail se veut la réalisation d'un langage de programmation appartenant au paradigme logique. La sémantique de l'article de référence tente de surmonter les difficultés dues à la négation en manipulant simultanément des valeurs pour lesquelles les prédicats peuvent être vrais et des valeurs pour lesquelles les prédicats peuvent être faux. La réalisation s'appuie sur un algorithme générique de type point fixe.

Abstract

This work is an attempt to create a language belonging to the logic programming. The semantic framework in the reference article tries to bridge the gap caused by difficulties due to negation in a way that manipulates values where predicates are true and values where predicates are false. The realization is based on a generic fixpoint algorithm.

Remerciements

C'est ici que survient la tâche la plus difficile, celle de la gratitude. Non que nous n'en ayons aucune, au contraire, mais bien qu'il nous faille user de toute notre délicatesse.

Nous tenons à remercier *Naji Habra*, notre promoteur, pour son optimisme permanent et communicatif. Nous tenons à remercier *Baudouin Le Charlier* pour des raisons tout à fait similaires. Nous tenons aussi à les remercier de la qualité et de l'intérêt de ce sujet de mémoire dont il en existe peu.

Sans le soutien de nos proches, par leur simple présence, dans la famille, dans les amis, ou sur les bancs de l'université, nous aurions peut-être trouvé la tâche plus ardue.

Ensuite, à la manière des deux mains dessinées par *M. C. Escher*, *Vincent* remercie *Tom*, et *Tom* remercie *Vincent*.

Mais, il y en a aussi sûrement d'autres, ...

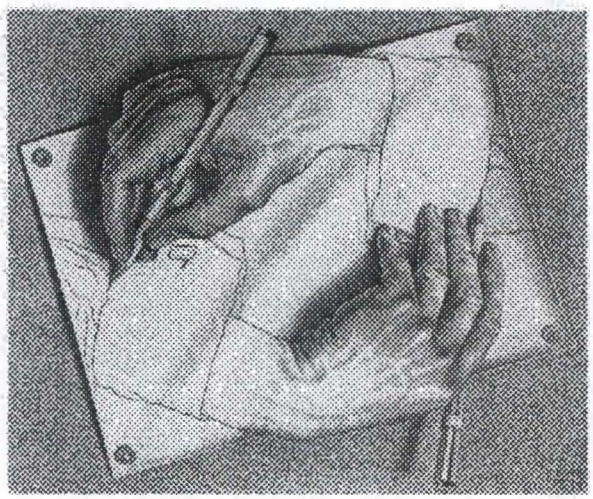


Table des matières

Introduction

La programmation logique reste un domaine ésotérique où les coûts de réalisation ne correspondent que très rarement aux attentes.

Sur base de deux articles principaux, ce travail modeste tente une approche dans ce domaine. Le premier article [4] fournit l'idée maîtresse du point de vue à adopter ici, à savoir une sémantique générale formelle. Tandis que le second [1] contribue à la réalisation de ce point de vue en fournissant un algorithme générique de calcul d'un point fixe. Ces deux briques de l'édifice constituent ensemble un résultat somme toute intéressant, bien que nous ne nous soyons intéressés qu'à une réalisation de certains aspects du problème global (paradigme prédicatif à domaines de valeurs finis).

Le langage le plus connu dans ce domaine est très certainement PROLOG qui, à l'heure d'aujourd'hui, a fait le tour du monde et dont il existe d'innombrables versions améliorées. Il souffre cependant de quelques boulets qu'il traîne à ses pieds :

- négation logique mal controlée ;
- difficulté d'appréhender des choses fausses.

Bien que l'on parvienne quand même à des résultats fantastiques avec cette base, d'autres chercheurs explorent des directions tout aussi originales que variées : c'est la cas de ce travail qui suit, dans l'ombre, les articles de référence.

L'effort de rédaction de ce document s'aligne dans le but de toucher pas mal de personnes, même celles qui ne regarde le paradigme de la programmation logique que de très loin. Sans échapper aux chapitres complexes du cœur du problème, nous avons voulu vulgariser quelques fonctionnements que nous avons mis en œuvre. Enfin, nous avons tenu à documenter quelque peu le petit programme que nous avons fait afin de faciliter la vie des générations futures qui désireraient grandement poursuivre la quête sur la même voie que celle que nous avons prise.

Ce travail ne se veut donc pas extraordinaire, il incarne substantifiquement une tentative et montre la complexité du domaine abordé, même si les bases théoriques restent quelque part très intéressantes et font preuve de performance.

Plan du rapport

Ce rapport consiste en cinq parties.

- Dans une première partie, nous rapelons quelques notions de base qui sont d'un intérêt général pour tout ce qui est la théorie de la programmation logique.
  - Le premier chapitre essaie en quelques pages d'introduire le lecteur au paradigme de la programmation logique. Nous proposons deux approches : d'abord l'approche classique (dite de PROLOG) et ensuite l'approche du β-langage, c'est à dire, celle que nous avons élaborée dans ce travail.
  - Dans un deuxième chapitre nous définissons les concepts de relation et de birelation ainsi que les opérations qu'on peut effectuer dessus.
  - Le chapitre suivant propose les concepts et les propriétés centraux de la théorie mathématique du point fixe. Comme il s'agît là du chapitre le plus mathématique de ce travail, nous avons pris soin d'introduire ce chapitre de la façon la plus intuitive possible. A une première lecture, la section introductive devrait suffire pour une bonne compréhension du reste du document.
  - Ayant expliqué de façon purement théorique ce que c'est qu'unpoint fixe, nous proposons au chapitre (4) un algorithme qui permet de le calculer efficacement. Ce chapitre n'est en fait rien qu'une autre présentation de l'algorithme proposé par [1].
- Dans la deuxième partie du rapport nous pouvons alors introduire le β-langage.
  Cette présentation est découpée en quatres parties :
  - La syntaxe concrète (chapitre 5), qui décrit les règles qui doivent être respectées pour qu'une chaînes de caractères puisse être considérée comme un β-programme, ainsi que les messages d'erreur auxquels l'utilisateur peut s'attendre s'il ne respecte pas ces règles
  - La sémantique (chapitre 7) qui décrit le sens ou le résultat d'un tel programme. La définition théorique de cette sémantique est complétée avec un exemple de calcul et une comparaison entre cette β-sémantique et celle de PROLOG.
  - La syntaxe abstraite (chapitre 6) qui fait le lien entre les deux.
  - A ces trois chapitres traditionels, nous avons ajouté un quatrième (chapitre 8 : Un paradoxe du β-langage), assez intéressant d'un point de vue théorique, mais un peu gênant pour le statut de notre langage comme langage de programmation.
    En faisant des tests, nous nous sommes en effet rendu compte que notre interpréteur ne donne pas toujours les résultats attendus (jusqu'à un point où l'on peut effectivement parler de paradoxe).
    Dans ce chapitre, nous proposons le programme β qui nous a permis de découvrir le problème et nous démontrons de façon théorique que ce problème n'est pas dû à notre implémentation, mais est inhérent à la β-sémantique.
    Sans doute les observations faites dans ce chapitre devraient-elles avoir des implications au niveau de la démarche à suivre par un programmeur en β-langage, mais nous n'avons plus eu le temps d'explorer cela.
  - Dans la troisième partie nous regardons d'un peu plus près comment nous avons implémenté notre interpréteur.

Après une présentation globale de l'architecture de notre logiciel, chacun des onze modules qui la compose, sera examiné plus en détail.

Les chapitres les plus intéressants sont sans doute les chapitres (10) et (11) (sur la représentation des relations et l'implémentation des opérations relationnelles) et le chapitre (18) (sur le calcul de la sémantique, c'est-à-dire le calcul d'un point fixe dans notre cas particulier).

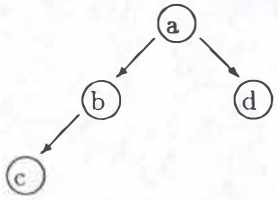
- La quatrième partie est allors dévouée à une petite présentation du résultat de notre travail. On explique comment invoquer notre interpréteur et on donne quelques petits exemples d'exécution.

  La liste des exemples sera complétée dans le chapitre (23.6) qui traite d'une application de notre programme à l'interprétation abstraite. Nous expliquons comment le β-langage peut être utile dans l'analyse de programmes PROLOG et nous présentons un petit programme (écrit en CaML) qui permet de générer, à partir d'un programme PROLOG, le programme en β-langage utilisable pour son analyse.
- Le travail est complété par le code source et un chapitre de conclusions.

1 Le paradigme logique

1.1 L'approche Prolog

Le langage de programmation logique PROLOG possède son approche propre tant du point de vue de la syntaxe d'un prédicat que de la façon de le résoudre. Nous allons examiner ici ces plans d'approche sans toutefois submerger le lecteur dans un détail outrancier.
Tout au long de ces approches, nous conserverons à l'esprit l'application suivante qui incarne un arbre généalogique où l'on ne répertorie que les individus de sexe masculin.



1.1.1 La syntaxe de PROLOG

La syntaxe de PROLOG pur est très simple. Ainsi, pour exprimer les faits de parenté, il nous faut énoncer que

- a est le père de b ;
- b est le père de c ;
- a est le père de d ;

Le prédicat logique qui synthétise ces informations peut s'écrire comme suit

$$Pere(X,Y) \Longleftrightarrow (X = a \wedge Y = b) \vee (X = b \wedge Y = c) \vee (X = a \wedge Y = d)$$

Il jouit d'une forme particulière : la forme disjonctive, c'est-à-dire qu'il correspond à la forme générique

$$\bigvee_{i=1}^{n} \left[\bigwedge_{j=1}^{m_i} P_{ij}(X_1^{ij}, \ldots, X_{p_{ij}}^{ij}) \right]$$

où

$$\begin{aligned}
P_{11}(X_1^{11}) &\Leftrightarrow X_1^{11} = a \\
P_{12}(X_1^{12}) &\Leftrightarrow X_1^{12} = b \\
P_{21}(X_1^{21}) &\Leftrightarrow X_1^{21} = b \\
P_{22}(X_1^{22}) &\Leftrightarrow X_1^{22} = c \\
P_{31}(X_1^{31}) &\Leftrightarrow X_1^{31} = a \\
P_{32}(X_1^{32}) &\Leftrightarrow X_1^{32} = d
\end{aligned}$$

PROLOG repose sur une rédaction propre des prédicats logiques en cette forme disjonctive. Il suffit alors d'aligner les différentes parties du prédicat en *clauses* possédant le même en-tête. Dans notre exemple, nous disposons de constantes a, b, c, ce qui permet d'écrire très synthétiquement :

```
Pere(a,b).
Pere(b,c).
Pere(a,d).
```

Cette relation de filiation peut être étendue en sautant les générations. Ainsi, un prédicat Ancetre(X,Y) peut être défini comme suit :

$$Ancetre(X,Y) \Longleftrightarrow Pere(X,Y) \vee (\exists Z \text{ t.q. } Pere(X,Z) \wedge Ancetre(Z,Y))$$

PROLOG transforme ceci en deux lignes, supposant implicitement la quantification existentielle :

```
Ancetre(X,Y) :- Pere(X,Y).
Ancetre(X,Y) :- Pere(X,Z), Ancetre(Z,Y).
```

En synthèse, on obtient une base de raisonnement grâce au prédicat Pere et des règles de déductions grâce au prédicat Ancetre.

```
Pere(a,b).
Pere(b,c).
Pere(a,d).
Ancetre(X,Y) :- Pere(X,Y).
Ancetre(X,Y) :- Pere(X,Z), Ancetre(Z,Y).
```

Ayant inscrit toutes ces informations dans un fichier, il devient possible d'utiliser le moteur de recherche de PROLOG afin de lui poser certaines questions. Par exemple :

?-Pere(a,b) ou ?-Pere(a,X) ou ?-Ancetre(X,Y).

Ce moteur de recherche travaille grâce à la technique très puissante des *substitutions*. Sans développer formellement cet objet, nous allons simplement expliquer ce qui se passe ainsi que les différentes étapes du processus d'interprétation dans notre exemple.

1.1.2 Détail de requêtes

Examinons la requête

?- Pere(b,c).

PROLOG va passer en revue toutes les déclarations ayant le même en-tête, selon l'ordre de rédaction du fichier source. D'abord il rencontre la clause spécifiant que a est le père de b, qu'il rejette puisqu'il ne s'agit pas de ce que l'on cherche. Ensuite, il rencontre la clause spécifiant que b est le père de c, auquel cas il écrit *"Yes"* à l'utilisateur car cela correspond bien à la requête. Quant à le troisième clause, elle ne donne rien, à l'instar de la première ; le prédicat Ancetre(·,·) n'est pas concerné.
Examinons la requête

Pere(A,B).

Dans ce cas, PROLOG écrit les trois liens de parenté qui existent dans notre exemple car A et B sont considérés comme des variables. Elles vont successivement recevoir les valeurs constantes spécifiées dans les énoncés des clauses, ce qui provoque l'affichage :
A=a, B=b ;
A=b, B=c ;
A=a, B=d ;
en respectant l'ordre de rédaction des clauses car le moteur examine le fichier du haut vers le bas strictement.
Examinons la requête

Ancetre(A,B).

PROLOG rencontre la première clause de ce prédicat :

Ancetre(X,Y) :- Pere(X,Y).

Ce qui lui permet de poser l'analogie[1] :
A$\longleftrightarrow$X
B$\longleftrightarrow$Y
Il va dès lors tenter une recherche avec Pere(A,B), et comme il est dit dans la clause, toute solution de cette nouvelle recherche sera une solution de la requête ?- Ancetre(A,B). Nous l'avons déjà fait, et nous trouvons :
A=a, B=b ;
A=b, B=c ;
A=a, B=d ;
La deuxième clause passe alors sur la sellette :

Ancetre(X,Y) :- Pere(X,Z), Ancetre(Z,Y).

[1] Cette analogie est en substance une substitution comme cela a été cité plus tôt. Bien qu'il en existe des cas plus complexes et mieux formalisés, il est suffisant ici de ne pas en dire plus car le but de ces préliminaires est de situer le lecteur dans un domaine qui lui est étranger.

Elle fait tenir la même analogie :
A$\longleftrightarrow$X
B$\longleftrightarrow$Y
PROLOG recherche alors des solutions pour la première partie Pere(A,C) pour laquelle il suspend la requête Ancetre(X,Y) et lance une résolution. Les solutions maintenant devenues classiques de Pere($\cdot$,$\cdot$) reviennent :
A=a, B=b ;
A=b, B=c ;
A=a, B=d ;

Plus précisément, ces trois solutions arrivent au compte-goutte car quand la première arrive, PROLOG la propage le plus loin possible et quand enfin il ne peut plus rien en tirer, il laisse venir la deuxième, puis la troisième. Pour ce voir, intercalons-nous sur l'échele du temps à l'instant où la première solution arrive (A=a, C=b). Le moteur est donc parvenu à identifier des valeurs pour X et Z, si l'on reprend les noms des variables originelles de la déclaration. Il va utiliser cette information en traitant la seconde partie de la clause avec Ancetre(Z,Y) grâce à l'analogie :
C$\longleftrightarrow$Z (*i.e. ici b*)
B$\longleftrightarrow$Y
pour repartir avec la question Ancetre(b,B). Il est alors nécessaire de passer en revue tous les prédicats du fichier source et en particulier les clauses propres au prédicat Ancetre($\cdot$,$\cdot$). La première clause renvoie
B=c ;
La seconde n'apporte aucune information.
En revenant de cette sous-requête, on peut ajouter une solution supplémentaire à la requête Ancetre(A,B) :
A=a, B=c ;
Aux autres instants de l'échelle du temps où les solutions A=b, C=c ; et A=a, C=d ; surviennent, il n'est pas utile de développer les événements car ces solutions ne viennent pas enrichir l'ensemble des solutions.
En synthèse, PROLOG a successivement trouvé :
A=a, B=b ;
A=b, B=c ;
A=a, B=d ;
A=a, B=c ;

1.1.3 Précautions

La façon dont nous avons rédigé nos prédicats en PROLOG n'a pas été choisie au hasard. Ce langage souffre en effet d'un forte dépendance vis-à-vis de l'ordre de la rédaction.
Ainsi notre programme

```
{ Ancetre(X,Y) :- Pere(X,Y).
  Ancetre(X,Y) :- Pere(X,Z), Ancetre(Z,Y).
```

ne donne pas les mêmes résultats s'il devient

```
{ Ancetre(X,Y) :- Pere(X,Y).
  Ancetre(X,Y) :- Ancetre(Z,Y), Pere(X,Z).
```

ou encore

```
{ Ancetre(X,Y) :- Pere(X,Z), Ancetre(Z,Y).
  Ancetre(X,Y) :- Pere(X,Y).
```

ni avec

```
{ Ancetre(X,Y) :- Ancetre(Z,Y), Pere(X,Z).
  Ancetre(X,Y) :- Pere(X,Y).
```

Cette difficulté d'écriture a apporté en son temps beaucoup de désillusions. Aujourd'hui encore, nombreux sont les débutants qui ne se rendent pas compte de ces subtilités et buttent sur un coût de mise en œuvre élevé.
Par ailleurs, notre exemple ne renferme aucune négation ! PROLOG souffre également de l'existence de négations et ne fonctionne adéquatement que dans certains cas précis où la prudence est de mise. L'idée de PROLOG est de considérer comme vrai ce qui peut être déduit du sytème logique spécifié dans le fichier source, et de considérer comme faux ce qui ne peut l'être. Divers travaux ont essayé de nuancer cette caricature mais elle se résume essentiellemnt à ce qui vient d'être dit. Pas question donc de chercher n'importe comment ce qui est faux par rapport au système logique spécifié.

1.2 L'approche du β-langage

Notre langage se base également sur la forme disjonctive dont nous avons parlé plus tôt :

$$\bigvee_{i=1}^{n} \left[\bigwedge_{j=1}^{m_i} P_{ij}(X_1^{ij}, \ldots, X_{p_{ij}}^{ij}) \right]$$

Toutefois, la multiplicité des en-têtes qui apparaissait dans PROLOG n'est plus : un seul en-tête suffit. Notre exemple généalogique prend la forme suivante :

```
Pere(X,Y)      :=  X=a & Y=b
                |  X=b & Y=c
                |  X=a & Y=d;
Ancetre(X,Y)   :=  Pere(X,Y).
                |  Pere(X,Z) & Ancetre(Z,Y).
```

Il s'agit bien sûr d'une pure convention d'écriture. Par contre, le β-langage résout les requêtes tout autrement que PROLOG. L'objectif du β-langage est d'emmagasiner les solutions d'un prédicat. En conséquence, l'ensemble des tuples (ici des couples) est scindé en deux parties :

- les tuples pour lesquels le prédicat est vrai ;
- les tuples pour lesquels le prédicat est faux.

Ces deux ensembles forment à eux deux ce que l'on appelle une *birelation* (voir point 2.4). Ainsi la partie dite positive (pour laquelle les tuples sont vrais) interprète le prédicat comme il est écrit dans le code, tandis que la partie négative de la birelation interprète le prédicat en permutant les rôles des conjonctions et disjonctions (voir chapitre 7).

$$\begin{array}{lcl}
\text{Pere}(X,Y) & \Longleftrightarrow & X{=}a \wedge Y{=}b \\
& \vee & X{=}b \wedge Y{=}c \\
& \vee & X{=}a \wedge Y{=}d\,;
\end{array}$$

$$\begin{array}{lcl}
\neg\text{Pere}(X,Y) & \Longleftrightarrow & X{\neq}a \vee Y{\neq}b \\
& \wedge & X{\neq}b \vee Y{\neq}c \\
& \wedge & X{\neq}a \vee Y{\neq}d\,;
\end{array}$$

$$\begin{array}{lcl}
\text{Ancetre}(X,Y) & \Longleftrightarrow & \text{Pere}(X,Y). \\
& \vee & \exists Z\ \text{Pere}(X,Z) \wedge \text{Ancetre}(Z,Y).
\end{array}$$

$$\begin{array}{lcl}
\neg\text{Ancetre}(X,Y) & \Longleftrightarrow & \neg\text{Pere}(X,Y). \\
& \wedge & \forall Z\ \neg\text{Pere}(X,Z) \vee \neg\text{Ancetre}(Z,Y).
\end{array}$$

Autrement dit, tout prédicat, selon qu'il est interprété positivement ou négativement, permet de cerner l'ensemble des tuples pour lesquels il est vrai ou faux.
Comme nous l'avons déjà dit, le β-langage utilise une technique différente de celle de PROLOG : on associe à chaque prédicat une birelation constituée de deux ensembles vides :

$$\text{départ} = (\langle Pos_{Pere}, Neg_{Pere}\rangle, \langle Pos_{Ancetre}, Neg_{Ancetre}\rangle) = (\langle\emptyset,\emptyset\rangle\langle\emptyset,\emptyset\rangle)$$

qui vont grossir au fur et à mesure de l'avancement de l'algorithme de résolution. Ainsi, toutes les birelations peuvent être manipulées simultanément, gonflant progressivement. Examinons maintenant les itérations.

Le point de départ

$Pos^0_{Pere} = \emptyset$	$Pos^0_{Ancetre} = \emptyset$
$Neg^0_{Pere} = \emptyset$	$Neg^0_{Ancetre} = \emptyset$

Ensuite, il s'agit de calculer $Pos^1_{Pere}, Neg^1_{Pere}, Pos^1_{Ancetre}, Neg^1_{Ancetre}$ à l'aide d'un règle de transformation

$$\begin{array}{l}
(\langle Pos^i_{Pere}, Neg^i_{Pere}\rangle, \langle Pos^i_{Ancetre}, Neg^i_{Ancetre}\rangle) \\
\quad \rightsquigarrow (\langle Pos^{i+1}_{Pere}, Neg^{i+1}_{Pere}\rangle, \langle Pos^{i+1}_{Ancetre}, Neg^{i+1}_{Ancetre}\rangle)
\end{array}$$

qui dans notre contexte impose les définitions suivantes :

$$
\begin{aligned}
Pos^{i+1}_{Pere}(X,Y) \quad \Longleftrightarrow \quad & X \in \{a\} \wedge Y \in \{b\} \\
\vee \quad & X \in \{b\} \wedge Y \in \{c\} \\
\vee \quad & X \in \{a\} \wedge Y \in \{d\}
\end{aligned}
$$

$$
\begin{aligned}
Neg^{i+1}_{Pere}(X,Y) \quad \Longleftrightarrow \quad & X \in \{b,c,d\} \vee Y \in \{a,c,d\} \\
\wedge \quad & X \in \{a,c,d\} \vee Y \in \{a,b,d\} \\
\wedge \quad & X \in \{b,c,d\} \vee Y \in \{a,b,c\}
\end{aligned}
$$

$$
\begin{aligned}
Pos^{i+1}_{Ancetre}(X,Y) \quad \Longleftrightarrow \quad & Pos^{i}_{Pere}(X,Y) \\
\vee \quad & \exists Z \ (Pos^{i}_{Pere}(X,Z) \wedge Pos^{i}_{Ancetre}(Z,Y))
\end{aligned}
$$

$$
\begin{aligned}
Neg^{i+1}_{Ancetre}(X,Y) \quad \Longleftrightarrow \quad & Neg^{i}_{Pere}(X,Y) \\
\wedge \quad & \forall Z \ (Neg^{i}_{Pere}(X,Z) \vee Neg^{i}_{Ancetre}(Z,Y))
\end{aligned}
$$

Première itération Après une première application de cette transformation, nous trouvons :

Pos^1_{Pere}	=	$\{(a,b),(b,c),(a,d)\}$
Neg^1_{Pere}	=	$\{(a,a),(a,c),(b,a),(b,b),(b,d),(c,a),$ $(c,b),(c,c),(c,d),(d,a),(d,b),(d,c),(d,d)\}$
$Pos^1_{Ancetre}$	=	$\emptyset$
$Neg^1_{Ancetre}$	=	$\emptyset$

$Pos^1_{Ancetre}$ et $Neg^1_{Ancetre}$ ne bougent pas car les règles de transformation réfèrent des ensembles vides. Il faut encore itérer une fois pour les voir évoluer.

Troisième itération On obtient :

Pos^1_{Pere}	=	$\{(a,b),(b,c),(a,d)\}$
Neg^1_{Pere}	=	$\{(a,a),(a,c),(b,a),(b,b),(b,d),(c,a),$ $(c,b),(c,c),(c,d),(d,a),(d,b),(d,c),(d,d)\}$
$Pos^1_{Ancetre}$	=	$\{(a,b),(b,c),(a,d)\}$
$Neg^1_{Ancetre}$	=	$\emptyset$

Quatrième itération On obtient :

Pos^1_{Pere}	=	$\{(a,b),(b,c),(a,d)\}$
Neg^1_{Pere}	=	$\{(a,a),(a,c),(b,a),(b,b),(b,d),(c,a),$ $(c,b),(c,c),(c,d),(d,a),(d,b),(d,c),(d,d)\}$
$Pos^1_{Ancetre}$	=	$\{(a,b),(b,c),(a,d),(a,c)\}$
$Neg^1_{Ancetre}$	=	$\{(b,a),(b,b),(b,d),(c,a),(c,b),(c,c),$ $(c,d),(d,a),(d,b),(d,c),(d,d)\}$

Cinquième itération On obtient :

Pos^1_{Pere}	=	$\{(a,b),(b,c),(a,d)\}$
Neg^1_{Pere}	=	$\{(a,a),(a,c),(b,a),(b,b),(b,d),(c,a),$ $(c,b),(c,c),(c,d),(d,a),(d,b),(d,c),(d,d)\}$
$Pos^1_{Ancetre}$	=	$\{(a,b),(b,c),(a,d),(a,c)\}$
$Neg^1_{Ancetre}$	=	$\{(a,a),(b,a),(b,b),(b,d),(c,a),(c,b),$ $(c,c),(c,d),(d,a),(d,b),(d,c),(d,d)\}$

Cette itération est la dernière. Toute autre itération du processus ne changera rien. On dit alors que le *point fixe* de la transformation est atteint. Ainsi, lors d'une requête, l'algorithme se met en marche jusqu'à l'obtention du point fixe du prédicat concerné.

On remarquera qu'il n'y a pas ici de dépendance croisée, c'est-à-dire de partie positive dépendant d'une partie négative ou inversément, car simplement, notre programme ne comporte aucune négation. Cela ne consitute toutefois aucun obstacle.

L'avantage principal sur PROLOG est double

- pas de problème de gestion de la négation puisque tous les tuples pour lesquels le prédicat est faux sont traités au fur et à mesure de l'avancement de l'algorithme ;
- pas de contrainte d'ordre des appels dans l'écriture puisque l'algorithme se base principalement sur les birelations associées aux prédicats et non sur la rédaction elle-même comme c'était le cas de PROLOG. Donc pas de bouclage infini.

2 Les relations

2.1 Notions de base

Définition 1 (Relation)
On définit une relation sur un ensemble $U_1 \times \ldots \times U_n$ *comme étant un ensemble de tuples* $t \in U_1 \times \ldots \times U_n$.

Dans le cadre de travail où nous nous trouvons, chaque espace U_i se voit associé à une étiquette ou label. A une relation est donc associée une famille de labels $(l_1, \ldots, l_n)$. On obtient alors une bijection entre les tuples et les tuples étiquetés qui sont des paires.

$$< a_1, \ldots, a_n > \longleftrightarrow < l_1 : a_1, \ldots, l_n : a_n >$$

Ceci provoque l'apparition de deux types de fonctions. Le premier est lié au domaine de la relation :

$$\begin{array}{rccc} \mathcal{L} : & L & \longrightarrow & \{U_1, \ldots, U_n\} \\ & l_i & \rightsquigarrow & U_i \end{array}$$

Le second regroupe beaucoup plus de fonctions. Il permet de voir un tuple sous un point de vue différent :

Définition 2 (Tuple étiqueté)
Un tuple étiqueté par $\{l_{i_1}, \ldots, l_{i_n}\}$ *est une fonction*

$$\begin{array}{rccl} t & : & \{l_{i_1}, \ldots, l_{i_n}\} & \rightarrow \bigcup_i U_i \\ & : & l_{i_k} & \mapsto t[l_{i_k}] \in U_{i_k}. \end{array}$$

Il sera noté sous la forme

$$t =< l_{i_1} : a_1, \ldots, l_{i_n} : a_n >$$

L'existence de ce second type de fonctions montre que tout tuple peut aussi se représenter sous la forme d'un tableau, et par suite, qu'un tableau peut synthétiser plusieurs tuples. Les titres des colonnes sont les étiquettes associées au domaine auquel le tuple appartient.
Quand le contexte le permet, c'est-à-dire assez souvent, on assimile implicitement un tuple à un tuple étiqueté et l'on sous-entend l'existence des fonctions citées.
Si nous considérons par exemple :

$U_1 = ensemble\ des\ prénoms$
$U_2 = ensemble\ des\ âges$
$l_1 = prénom$
$l_2 = âge$

on peut obtenir entre autre

prénom	*âge*
"Tom"	25
"Vincent"	23
"Toto"	37
"Nabuchodonosor"	69
"Mathusalem"[2]	969

Les fonctions évoquées se décrivent aisément :

$$\begin{array}{rccc} \mathcal{L} : & \{prénom, âge\} & \longrightarrow & \{ensemble\ des\ prénoms,\ ensemble\ des\ âges\} \\ & prénom & \rightsquigarrow & ensemble\ des\ prénoms \\ & âge & \rightsquigarrow & ensemble\ des\ âges \end{array}$$

et sans être exhaustif, on citera les deux fonctions suivantes :

$$\begin{array}{rccc} t_{("Tom",25)} : & \{prénom, âge\} & \longrightarrow & \{prénoms\} \cup \{âges\} \\ & prénom & \rightsquigarrow & "Tom" \\ & âge & \rightsquigarrow & 25 \end{array}$$

$$\begin{array}{rccc} t_{("Mathusalem",969)} : & \{prénom, âge\} & \longrightarrow & \{prénoms\} \cup \{âges\} \\ & prénom & \rightsquigarrow & "Mathusalem" \\ & âge & \rightsquigarrow & 969 \end{array}$$

Il existe donc une autre bijection entre ces fonctions et les tuples de la relation.
Par suite, on étend la notion de relation à celle de relation étiquetée :

[2] Genèse 5,27. En anglais, il est écrit *And all the days of Methuselah were nine hundred sixty and nine years : and he died.*

Définition 3 (Relation étiquetée)

Une relation étiquetée par $\{l_{i_1}, \ldots, l_{i_n}\}$ *est une ensemble de tuples étiquettés par* $\{l_{i_1}, \ldots, l_{i_n}\}$.

En plus de ceci, il existe deux relations particulières, étiquetées par l'ensemble vide et notées comme TRUE *et* FALSE.

Définition 4

Nous noterons l'ensemble des relations étiquetées comme $\mathcal{SR}$[3].

2.2 Définitions et propriétés

Théorème 1 (Théorème du berger)

Soient A et B deux ensembles finis, et #A et #B désignant respectivement le nombre d'éléments que possède A et le nombre d'éléments que possède B,
Alors le nombre d'éléments que possède le produit cartésien $A \times B$ *de ces deux ensembles est*

$$\#(A \times B) = \#A \cdot \#B$$

La conséquence immédiate de ce théorème anodin est la croissance rapide du nombre de tuples possibles. Il devient dès lors très vite impossible d'énumérer exhaustivement tous les tuples appartenant à une relation donnée. D'autres représentations doivent apparaître, plus succintes.
L'économie de représentation qui est réalisée est basée sur le fait que de nombreux tuples peuvent avoir une composante commune.
Nous allons à présent étaler quelques définitions qui s'enchaînent aisément. La première (restriction d'une relation) correspond à une opération de projection.

Définition 5 (Restriction d'une relation)

Soit R une relation étiquetée par $\{l_1, \ldots, l_n\}$. *La restriction de R par rapport à l'étiquette* l_k ($k \in \{1, \ldots, n\}$) *est la relation notée*

$$\mathcal{R}est_{l_k} R$$

étiquetée par $\{l_1, \ldots, l_{k-1}, l_{k+1}, \ldots, l_n\}$ *et telle que :*

$$\mathcal{R}est_{l_k} R = \{< l_1 : x_1, \ldots, l_{k-1} : x_{k-1}, l_{k+1} : x_{k+1}, \ldots, l_n : x_n > |$$
$$\exists x \in U_{i_k} :< l_1 : x_1, \ldots, l_{k-1} : x_{k-1}, l_k : x, l_{k+1} : x_{k+1}, \ldots, l_n : x_n > \in R.\}$$

La restriction d'une relation sur une étiquette revient à projecter tous les tuples en ne considérant pas cette étiquette donnée.

Définition 6 (Une étiquette saturée)

Soit R une relation étiquetée par $\{l_{i_1}, \ldots, l_{i_n}\}$. *On appelle* l_k ($k \in \{1, \ldots, n\}$) *une étiquette saturée de R si :*

$$\forall x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n :$$
$$< l_1 : x_1, \ldots, l_{k-1} : x_{i-1}, l_{k+1} : x_{i+1}, \ldots, l_n : x_n > \in \mathcal{R}est_{l_i}(R)$$
$$\Rightarrow \forall x \in U_k :< l_1 : x_1, \ldots, l_{k-1} : x_{k-1}, l_k : x, l_{k+1} : x_{k+1}, \ldots, l_n : x_n > \in R.$$

Définition 7 (Réduction simple)

R' est une réduction simple de R par rapport à l'étiquette l_k

$$\overset{not}{\Longleftrightarrow} R' = \mathcal{R}ed_{\{l_k\}}(R)$$
$$\overset{def}{\Longleftrightarrow}$$

$$\begin{array}{lll} & l_k \text{ est une étiquette saturée de } R & \wedge\ R' = \mathcal{R}est_{l_k} R \\ \vee & l_k \text{ n'est est pas une étiquette saturée de } R & \wedge\ R' = R \end{array}$$

[3] *Set of Relations.*

Définition 8 (Relation vraie partout)

On dira que TRUE est la relation simplement réduite de R si R est une relation dont le domaine est le produit cartésien à une composante U et si $R = U$.

Définition 9 (Relation fausse partout)

On dira que FALSE est la relation simplement réduite de R si R est une relation dont le domaine est le produit cartésien à une composante U et si $R = \emptyset$.

Cette notion de réduction simple porte sur un label donné. Elle peut très facilement être généralisée à plusieurs labels.

Définition 10 (Réduction)

R' est une réduction de R selon les étiquettes $l_{i_1}, \ldots, l_{i_k}$
$\stackrel{def}{\Longleftrightarrow} R' = \mathcal{R}ed_{\{l_{i_1}\}}(\mathcal{R}ed_{\{l_{i_2}\}}(\ldots \mathcal{R}ed_{\{l_{i_k}\}}(R)))$

Il est aussi possible de pousser la réduction jusqu'à son comble en procédant par tous les labels.

Définition 11 (Réduction maximale)

$R^\star$ est une réduction maximale de la relation R
$\stackrel{def}{\Longleftrightarrow}$

$$R^\star = \mathcal{R}ed_{l_1}(\ldots(\mathcal{R}ed_{l_n}(R)))$$

Définition 12 (Relations réduites équivalentes)

Deux relations sont dites équivalentes si elles ont même réduction maximale, c'est-à-dire si

$$R^\star = S^\star$$

On désigne par $\mathcal{R}ed^\star(R)$ l'ensemble de toutes les relations équivalentes à R.

Corollaire 1

Soit R une relation telle que R couvre son domaine tout entier,
Alors on a que

$$\mathcal{R}ed^\star(R) \ni TRUE, \text{ i.e. } R^\star = TRUE$$

Soit R une relation telle que R ne contient aucun élément de son domaine,
Alors on a que

$$\mathcal{R}ed^\star(R) \ni FALSE, \text{ i.e. } R^\star = FALSE$$

Pour illustrer cela, nous pouvons nous placer dans le contexte suivant :

$$L = \{X, Y, Z\}$$

$$\mathcal{D} = \{U_1, U_2, U_3\} = \{\{a, b\}, \{0, 1, 2\}, \{p, q\}\}$$

$$\begin{array}{lccc} \mathcal{L}: & L & \longrightarrow & \mathcal{D} \\ & X & \rightsquigarrow & \{a, b\} \\ & Y & \rightsquigarrow & \{0, 1, 2\} \\ & Z & \rightsquigarrow & \{p, q\} \end{array}$$

On peut avoir par exemple la relation suivante :

$$R_1 = \begin{array}{|c|c|c|} \hline X & Y & Z \\ \hline a & 0 & p \\ b & 0 & p \\ a & 2 & q \\ b & 2 & q \\ \hline \end{array} = \{< a, 0, p >, (b, 0, p), (a, 2, q), (b, 2, q)\}$$

qui se ramène à (réduction simple sur le label X)

$$R'_1 = \begin{array}{|c|c|}\hline Y & Z \\ \hline 0 & p \\ 2 & q \\ \hline \end{array} \in \mathcal{R}ed^\star(R_1)$$

Un autre exemple est

$$R_2 = \begin{array}{|c|c|}\hline X & Z \\ \hline a & p \\ a & q \\ b & p \\ b & q \\ \hline \end{array} \quad \text{qui devient} \longrightarrow \quad R'_2 = TRUE \in \mathcal{R}ed^\star(R_2)$$

Corollaire 2 (Classe d'équivalence)

L'opération $\mathcal{R}ed^\star(\bullet)$ définit des classes d'équivalences sur l'espace des relations finies. Ceci vient du fait que

$$R\rho S \iff S \in \mathcal{R}ed^\star(R) \vee R \in \mathcal{R}ed^\star(S) \iff R^\star = S^\star$$

définit une relation d'équivalence entre les relations (réflexive, symétrique, transitive). L'élément réduit maximal associé à une relation étant unique, on peut caractériser une classe d'équivalence par l'élément maximal qui lui en est déduit.

2.3 Opérations sur les relations

2.3.1 Intersection

Pour deux relations de même domaine, *i.e.* $R_1, R_2 \subseteq U_1 \times \ldots \times U_n$, on définit très facilement l'intersection de deux relations en passant par l'intersection ensembliste des ensembles de tuples qui définissent exhaustivement les deux relations. Cette observation est la base du raisonnement.
Toutefois, quelques uns des paramètres exposés plus tôt ont été négligés :

- omission des labels ;
- permutation de deux espaces identiques possibles dans les produits cartésiens des domaines ;
- les relations peuvent être des réductions de deux autres relations ;
- etc.

Le cas élémentaire traité ci-avant suppose implicitement que les labels associés aux deux relations sont les mêmes et que les fonctions $\mathcal{L}_{R_1}$ et $\mathcal{L}_{R_2}$ sont aussi les mêmes.
Pour définir plus précisément et formellement l'opération d'intersection, nous allons utiliser une opération d'*élévation* qui peut être assimilée au contraire d'une réduction.

Définition 13 (Elévation)
Soit R, une relation de domaine $D = U_1 \times \ldots \times U_p$ et de famille de labels $L = (l_1, \ldots, l_p)$. Soit également $M = (m_1, \ldots, m_q)$ une autre famille de labels, ainsi que les domaines $\mathcal{D} = \{V_1, \ldots, V_q\}$. On appelle élévation de R par rapport aux labels M et au domaine $\mathcal{D}$ la relation

$$S \stackrel{not}{=} Elev(R, M, \mathcal{D})$$

telle que

$$S^\star = R^\star$$

et S est une relation saturée sur labels $\{m_1, \ldots, m_q\}$.

La nouvelle relation S possède dès lors une fonction $\mathcal{L}_S$ définie comme suit :

$$\begin{array}{rccl} \mathcal{L}_S : & L \cup M & \longrightarrow & \{U_1, \ldots, U_p\} \bigcup \{V_1, \ldots, V_q\} \\ & l & \rightsquigarrow & \mathcal{L}_S(l) = \left\{ \begin{array}{ll} U_k & \text{si } l = l_k \in L \\ V_k & \text{si } l = m_k \in M \end{array} \right. \end{array}$$

et

$$t \in S \iff \exists u \in R \text{ tel que } t_{|L} = u$$

On remarquera qu'il n'est pas possible de procéder à une élévation s'il y a un conflit entre les domaines associés aux labels, c'est-à-dire quand

$$\exists l_k \ \exists m_p \text{ t.q. } U_k \neq V_p.$$

car la fonction $\mathcal{L}_S$ n'est pas définissable. Dans l'algorithme de notre β-langage, ce problème est contourné par une opération substituant des labels (voir section 2.3.8).

Corollaire 3

R est une réduction de S par rapport aux labels $(m_1, \ldots, m_q)$.

Cette définition permet la formalisation de l'opération d'intersection. En effet,

Soient R_1 de domaine $\mathcal{D} = U_1 \times \ldots \times U_n$
de labels $L_{R_1} = (l_1, \ldots, l_n)$
de fonction $\mathcal{L}_{R_1}$
et R_2 de domaine $\mathcal{E} = T_1 \times \ldots \times T_m$
de labels $L_{R_2} = (m_1, \ldots, m_m)$
de fonction $\mathcal{L}_{R_2}$

Sur cette base, on définit

$$\begin{aligned} C &= L_{R_1} \cap L_{R_2} \\ S_1 &= Elev(R_1^{L_{R_1}}, L_{R_2} \setminus C, \{T_1, \ldots, T_m\} \setminus \{U_1, \ldots, U_n\}) \\ S_2 &= Elev(R_2^{L_{R_2}}, L_{R_1} \setminus C, \{U_1, \ldots, U_n\} \setminus \{T_1, \ldots, T_n\}) \end{aligned}$$

On a alors que

$$\forall l \in L_{R_1} \cup L_{R_2} \quad \mathcal{L}_{S_1}(l) = \mathcal{L}_{S_2}(l)$$

et aussi que $R_1 \in \mathcal{R}ed^*(S_1)$, $R_2 \in \mathcal{R}ed^*(S_2)$. Ces deux nouvelles relations ont le même domaine de définition qui est

$$\prod_{l \in L_{R_1} \cup L_{R_2}} \mathcal{L}_{S_1}(l) \stackrel{not}{=} \mathcal{F}$$

L'intersection apparaît alors comme

$$\begin{aligned} R &= \{t \in \mathcal{F} \dagger t \in S_1 \wedge t \in S_2\} \\ &\stackrel{i.e.}{=} Elev(R_1^{L_{R_1}}, L_{R_2} \setminus L_{R_1}, \{T_1, \ldots, T_m\} \setminus \{U_1, \ldots, U_n\}) \\ &\quad \bigcap Elev(R_2^{L_{R_2}}, L_{R_1} \setminus L_{R_2}, \{U_1, \ldots, U_n\} \setminus \{T_1, \ldots, T_n\}) \end{aligned}$$

En synthèse, il suffit de grossir les relations en vue de les réexprimer sur un seul et même domaine commun, puis de procéder enfin à l'intersection au sens ensembliste du terme.
Dans l'article de référence, cette opération s'appelle *Natural Join* (voir [4]).

2.3.2 Réunion

Le procédé reste essentiellement le même. La conclusion de ce raisonnement similaire mène à

$$\begin{aligned} R &= \{t \in \mathcal{F} \dagger t \in S_1 \vee t \in S_2\} \\ &= Elev(R_1^{L_{R_1}}, L_{R_2} \setminus L_{R_1}, \{T_1, \ldots, T_m\} \setminus \{U_1, \ldots, U_n\}) \\ &\quad \bigcup Elev(R_2^{L_{R_2}}, L_{R_1} \setminus L_{R_2}, \{U_1, \ldots, U_n\} \setminus \{T_1, \ldots, T_n\}) \end{aligned}$$

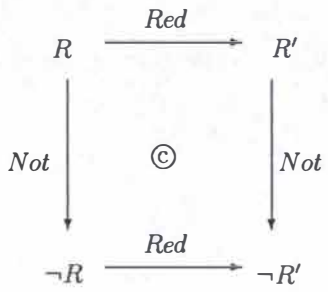
Dans l'article de référence, cette opération s'appelle *Conjoint Union* (voir [4]) .

2.3.3 Négation

Soit R une relation de labels $(l_1, \ldots, l_k)$ et de domaine $U_1 \times \ldots \times U_n \stackrel{not}{=} \mathcal{D}$. On définit la négation par

$$\neg R \stackrel{def}{=} \{t \in \mathcal{D} \mid t \notin R\}$$

Du point de vue des réductions, cela ne pase pas de problème. L'opération de réduction jouit d'une compatibilité vis-à-vis de la négation. Soient donc R et R' telles que $R' \in \mathcal{R}ed^*(R)$ et R plus générale que R' ; le diagramme suivant est commutatif :



2.3.4 Différence

Cette opération se définit comme

$$R \setminus S = R \bigcap \neg S$$

2.3.5 Projection

Cette opération consiste à restreindre une relation mais dans une optique différente de celle de la réduction. Soit R une relation de domaine $\mathcal{D} = U_1 \times \ldots \times U_n$ et de labels $L = (l_1, \ldots, l_n)$. Soit $P \subseteq L$, un ensemble de labels choisis dans L. La projection de la relation R sur P consiste en un ensemble de tuples pris dans R dont on n'a conservé que les composantes en rapport avec les labels contenus dans P. Le lecteur retrouvr dans ceci la définition de *restriction* (voir définition 5 page 16).
Ainsi,

$$\Pi_P R = \{t \in \mathcal{D}_{|P} \mid \exists t' \in R \; t'_{|P} = t\} = \mathcal{R}est_P(R)$$

La fonction associant les domaines aux labels est :

$$\begin{array}{rccc} \mathcal{L}_{\Pi_P R} : & P & \longrightarrow & \bigcup_{l \in P} \mathcal{L}_R(l) \\ & l & \rightsquigarrow & \mathcal{L}_R(l) \end{array}$$

2.3.6 Quantification existentielle

Une quantification existentielle se rapporte toujours à un label donné, et par suite à une famille de labels. De plus, elle diffère de l'opération de projection car au lieu de récolter les valeurs des tuples associés au label, elle les élimine, provoquant ainsi un rétrécissement du domaine de la relation.
Soit R une relation de domaine $\mathcal{D}$ et de labels L. Soit $P \subseteq L$, un ensemble de labels de L. Alors la nouvelle relation est

$$R' = \Pi_{P \setminus L}(R) = \mathcal{R}\text{est}_{P \setminus L}(R)$$

Cette opération singulière peut s'expliquer très simplement par le fait que lors d'une requête logique comme

$$\exists x \; p(x, y, z)$$

la valeur de x en elle-même n'a pas énormément d'importance. Il suffit conserver de cette recherche les valeurs de y et z pour lesquelles il sera possible de trouver une valeur vraie pour le prédicat $p(\bullet, \bullet, \bullet)$.
Sans que cela porte atteinte à ce qui vient d'être développé, l'ensemble P peut contenir des labels ne figurant pas dans l'ensemble de départ L. On peut sans remords ni regrets les enlever de P car il ne constituent aucune information pertinente. En effet, pour mettre l'ensemble des labels en synchronisation avec le domaine, il faut procéder à une élévation qui grossirait R en complétant les tuples par toutes les valeurs possibles présentes dans les domaines associés aux labels excédentaires (élévation). Procéder

ensuite à une projection de la nouvelle relation obtenue met immédiatement en évidence le côté inutile de l'élévation qui a précédé.

2.3.7 Quantification universelle

Cette opération ne peut s'exprimer en termes de projection comme cela est le cas de la quantification existentielle. Elle demande que l'on spécifie également un ensemble de labels sur lequel porte la quantification. Il s'agit ici de récolter les composantes des tuples en rapport avec les labels autres que ceux spécifiés, et ce en vérifiant qu'un label spécifié peut prendre dans la relation toutes les valeurs du domaine associé. On obtient :

$$R' = \{t \in \mathcal{D} \mid \forall x_1 \in U_{k_1}, \ldots, \forall x_p \in U_{k_p},\ \exists t' \in R \quad \text{t.q.} \quad t'_{|\{l_{k_1},\ldots,l_{k_p}\}} = (x_1, \ldots, x_p) \\ \wedge \quad t'_{|L\setminus\{l_{k_1},\ldots,l_{k_p}\}} = t\}$$

2.3.8 Substitution

La dernière opération relationnelle que nous devons considérer est la substitution. Elle permet de substituer les étiquettes d'une relation, soit pour d'autres étiquettes, soit pour des constantes.

Définition 14 (Une substitution)

Soit R une relation étiquetée par $\{l_{i_1}, \ldots, l_{i_n}\}$.

Une substitution pour R est une fonction

$$\sigma : \{l_{i_1}, \ldots, l_{i_n}\} \rightarrow L \cup [\cup_i U_i]$$

qui associe à chaque étiquette l_{i_k}

- *soit une constante $c_k \in U_{i_k}$*
- *soit une étiquette $l_{i_{k'}}$, telle que $U_{i_{k'}} = U_{i_k}$.*

Définition 15 (Le résultat d'une substitution)

Soit R une relation étiquetée par $\{l_{i_1}, \ldots, l_{i_n}\}$.
Soit σ une substitution pour R.

Soit

$$\{l_{j_1}, \ldots, l_{j_m}\} = \{\sigma(l_{i_k}) \mid k \in \{1, \ldots, n\} \wedge \sigma(l_{i_k}) \text{ est une étiquette}\}.$$

Le résultat de la substitution σ sur R,

$$subst_\sigma\ R$$

est l'ensemble de tuples

$$< l_{j_1} : r_1, \ldots, l_{j_m} : r_m >,$$

tels que la relation R contienne le tuple

$$< l_{i_1} : x_1, \ldots, l_{i_n} : x_n >,$$

où pour chaque $k \in \{1, \ldots, n\}$, la constante x_k est définie comme suit :

- *Si $\sigma(l_{i_k})$ est une constante c, alors $x_k = c$.*
- *Si $\sigma(l_{i_k})$ est une étiquette l_{j_l}, alors $x_k = r_l$.*

Terminons ce paragraphe avec quelques exemples appliqués à la relation suivante :

X	Y	Z
a	a	a
a	a	b
a	a	c
a	b	b
a	b	c
b	c	a

- La substitution $[X \rightarrow a]$ ne considère que les lignes pour lesquelles la valeur dans la colonne X est égale à a, pour ensuite enlever cette colonne.
  Ceci résulte à :

Y	Z
a	a
a	b
a	c
b	b
b	c

- La substitution $[X \rightarrow c]$ résulte de la même façon à :

  FALSE

- Si on applique la substitution $[Y \rightarrow Z]$, deux colonnes dans le tableau auront la même étiquette. On ne considère donc que les lignes pour lesquelles les valeurs dans ces deux colonnes sont égales. On trouve ainsi :

X	Z
a	a
a	b

- Si on applique la substitution $[Y \rightarrow Z, Z \rightarrow a]$, on ne considère que les lignes pour lesquelles la valeur dans la troisième colonne est égale à a, pour ensuite enlever cette colonne.
  Il ne reste alors qu'a remplacer l'étiquette Y par Z.

X	Z
a	a
b	c

- Remarquons que le résultat obtenu au point précédent diffère de ce qu'on obtient si on applique les deux substitutions $[Y \rightarrow Z]$ et $[Z \rightarrow a]$, l'une après l'autre.

  Le résultat obtenu en ce cas est en effet :

X	Z
a	a

  On voit donc qu'il est important d'effectuer toute la substitution d'un coup et pas terme par terme.

2.4 Birelations

Dans le langage implémenté pour ce travail de fin d'études, on ne travaille pas avec des relations mais des *birelations*. Le but est d'éviter certains inconvénients inhérents aux langages logiques ou prédicatifs classiques qui souffrent à manipuler les négations logiques et les prédicats faux. L'idée théorique de l'article de référence [4] est d'associer à chaque prédicat un ensemble formé de deux relations qui caractérisent repectivement ce qui est connu comme vrai pour le prédicat et ce qui est connu comme faux pour le prédicat. Les langages développés dans le passé ne se souciaient généralement que de la première partie, et en conséquence trouvaient une pierre d'achoppement en la négation logique. Cette idée de birelation se formalise comme suit :

Définition 16 (Birelation)

On appelle birelation sur $U_1 \times \ldots \times U_n$ *un couple* $< Pos, Neg >$ *tel que* $Pos \subseteq U_1 \times \ldots \times U_n$, $Neg \subseteq U_1 \times \ldots \times U_n$ *et* $Pos \bigcap Neg = \emptyset$.

Définition 17 (Ensemble des birelations)

On note l'ensemble des birelations par $\mathcal{SB}$[4].

[4] *Set of Birelations.*

3 La théorie mathématique du point fixe

3.1 Introduction

L'utilisateur de notre langage va nous proposer ce qui est essentiellement un système d'équations logiques que notre interpréteur devra résoudre.

C'est au chapitre 7 sur la sémantique du β-langage que nous allons rigoureusement spécifier ce que l'on entend par la solution d'un tel système d'équations logiques.
Mais, avant de faire cela, nous allons d'abord dans ce chapitre-ci, prendre un peu de distance par rapport à notre problème concret et considérer, dans un cadre beaucoup plus général, ce qu'est la solution d'un système d'équations et développer les outils mathématiques nécessaires à sa résolution.

Pour bien se fixer les idées, considérons le petit exemple suivant :

On a un domaine $U = \{a, b, c, d\}$ et deux ensembles $A, B \subseteq U$, qui satisfont le système d'équations suivant :

$$\begin{cases} A &= \{a\} \cup B \\ B &= \{b\} \cup A \end{cases} \tag{1}$$

On aimerait bien trouver la solution de ce système.
Bien que ce problème semble tout à fait trivial, plusieurs questions s'imposent :

- D'abord : Qu'entend-on exactement par la solution ?
  En effet, on trouve facilement qu'il y a plusieurs solutions possibles :

  $$A = \{a, b\} \qquad B = \{a, b\} \tag{2}$$
  $$A = \{a, b, c\} \qquad B = \{a, b, c\} \tag{3}$$
  $$A = \{a, b, c, d\} \qquad B = \{a, b, c, d\} \tag{4}$$

  On aurait envie de dire que la première solution est la solution du problème, en effet, elle a une position tout à fait particulière : d'une certaine manière, elle est la plus petite solution, les autres étant obtenues en faisant un choix arbitraire.
- Deuxièmement : Le système (1) est d'une telle symplicité que l'on voit, au premier coup d'œil, que la solution existe. Pour des systèmes plus compliqués, tel par exemple :

  $$\begin{cases} A &= \{a\} \cup B \cap C \\ B &= A \cup C \cap B \\ C &= \{b\} \cup B \cup C \end{cases}$$

  ceci n'est plus si évident que ça. N'existerait-il donc pas un critère général qui nous dit quand la plus petite solution existe ? Et, n'existerait-il pas un algorithme pour obtenir cette solution ?

La théorie mathématique et les définitions correpondantes que nous allons développer ci-dessous permettront exactement de répondre à ces questions.

Dans le cadre de cette théorie on va considérer :

- Un ensemble E, tel que la solution doive être trouvée parmi ses éléments.
- Une relation d'ordre $\sqsubseteq$ sur E, telle que pour deux propositions de solution $x, y \in E$, $x \sqsubseteq y$ veut dire que x est plus *générale*, ou contient moins d'information, que y.
- Un élément $\bot \in E$, qui est en effet l'élément le plus général, donc avec le moins d'information de E, et qui sera utilisé comme toute première approximation de la solution recherchée.
- Un opérateur $T : E \to E$, tel que si $x \in E$ est une approximation de la solution, $T(x)$ en est une qui est plus précise.
  Cet opérateur implémente d'une certaine manière les équations du système.

Ainsi, dans notre exemple (1) :

- On cherche une solution (A, B) dans l'ensemble

  $$E = U \times U$$

- La relation d'ordre sur E est définie par

$$(X_1, X_2) \sqsubseteq (Y_1, Y_2) \Leftrightarrow X_1 \subset Y_1 \wedge X_2 \subset Y_2,$$

de sorte que l'on puisse dire que la solution (2) est plus plus générale que (3) et (4).

- L'élément le plus général de E pour l'ordre considéré est bien évidemment

$$\bot = (\emptyset, \emptyset).$$

- L'opérateur T qui traduit le système d'équations est défini comme :

$$T(A, B) = (A', B')$$
$$\text{où} \begin{cases} A' = \{a\} \cup B \\ B' = \{b\} \cup A \end{cases}$$

La théorie mathématique que nous allons développer ci-dessous va en fait formuler des conditions générales sur la relation $\sqsubseteq$ et l'opérateur T[5], telles que la plus petite solution existe et puisse être obtenue comme la limite de la séquence

$$\bot,\ T(\bot),\ T(T(\bot)),\ \ldots$$

3.2 Les treillis complets

Définition 18 *Soit E un ensemble. On appelle une relation $R \subseteq E \times E$, un **ordre partiel** sur E ssi*

- *elle est réflexive :*

$$\forall x \in E : R(x, x)$$

- *elle est anti-symétrique :*

$$\forall x, y \in E : R(x, y) \wedge R(y, x) \Rightarrow x = y$$

- *elle est transitive :*

$$\forall x, y, z \in E : R(x, y) \wedge R(y, z) \Rightarrow R(x, z)$$

Définition 19 *Un **ensemble ordonné** (E, R) est composé d'un ensemble E et d'un ordre partiel R sur E.*

Rem. Un ordre est souvent noté de façon infixée, c'est-à-dire : on note $x \leq y$ ou $x \sqsubseteq y$ au lieu de $R(x, y)$.

Définition 20 *Soit $(E, \leq)$ un ensemble ordonné.*
*Un élément $a \in E$ est **une borne supérieure** d'un sous-ensemble $S \subseteq E$, si $\forall x \in S : x \leq a$.*
*Un élément $a \in E$ est **une borne inférieure** d'un sous-ensemble $S \subseteq E$, si $\forall x \in S : a \leq x$.*

Définition 21 *Soit $(E, \leq)$ un ensemble ordonné.*
*Un élément $a \in E$ est **la borne supérieure** d'un sous-ensemble $S \subseteq E$ (ce qu'on note par $a = \sqcup S$), si a et la plus petite des bornes supérieures de S, c'est-à-dire a est une borne supérieure de S, et pour chaque borne supérieure a' de S on a que $a \leq a'$.*
*Un élément $a \in E$ est **la borne inférieure** d'un sous-ensemble $S \subseteq E$ (ce qu'on note par $a = \sqcap S$), si a et la plus grande des bornes inférieures de S, c'est-à-dire a est une borne inférieure de S, et pour chaque borne inférieure a' de S on a que $a' \leq a$.*

On peut aisément démontrer que si la borne supérieure (resp. inférieure) d'un sous-ensemble $S \subseteq E$ existe, elle est unique.

Définition 22 *Un ensemble ordonné $(E, \leq)$ est **un treillis complet** si $\sqcup S$ existe pour chaque sous-ensemble $S \subseteq E$.*

Proposition 1 *Soit $(E, \leq)$ un treillis complet, alors $\sqcap S$ existe pour chaque sous-ensemble $S \subseteq E$.*

Démonstration : L'idée derrière la démonstration est que la plus grande borne inférieure d'un ensemble S et tout simplement la borne supérieure de toutes ses bornes inférieures.
Soit donc S^-, l'ensemble des bornes inférieures de S et $a = \sqcup S^-$.
Alors a est la borne inférieure de S, en effet :

[5] E muni de $\sqsubseteq$ doit être un treillis complet et T doit être continu

- a est une borne inférieure de S puisque chaque $y \in S$ est plus grand que tous les éléments de S^-, donc une borne supérieure de S^- et comme a est, par sa définition, la plus petite de ces bornes, $a \leq y$.
- Soit a' une autre borne inférieure de S, alors, par la définition de S^-, $a' \in S^-$ et, comme a est une borne supérieure de cet ensemble : $a' \leq a$.

Définition 23 *Soit $(E, \leq)$ un treillis complet.*
On note :
$\top = \sqcup E$, *le* ***top*** *de E,*
$\bot = \sqcap E$, *le* ***bottom*** *de E,*
Ces éléments sont caractérisés par la propriété suivante :
$\forall a \in E : \bot \leq a \leq \top$

Les seuls treillis complets que nous rencontrerons dans le cadre de ce travail sont décrits dans les deux propositions suivantes :

Proposition 2 *Soit A un ensemble, $\mathcal{P}(A)$ l'ensemble des sous-ensembles de A et $\subseteq$ la relation d'inclusion sur $\mathcal{P}(A)$,*
alors le couple $(\mathcal{P}(A), \subseteq)$ est un treillis complet.

Démonstration :

- Il est bien connu que $\subseteq$ est une relation d'ordre
- Soit $S \subseteq \mathcal{P}$ un ensemble de sous-ensembles.
  Considérons
  $$B = \bigcup_{X \in S} X,$$
  l'union des ensembles dans S. Alors $B = \sqcup S$, en effet :
  - B est une borne supérieure de S puisque $\forall X \in S : X \subseteq B$.
  - Soit B' une autre borne supérieure de S, alors par définition de la borne supérieure :
    $$\forall X \in S : X \subseteq B',$$
    de sorte que l'on a également que :
    $$\bigcup_{X \in S} X \subseteq B'$$
    et donc : $B \subseteq B'$.

Proposition 3 *Soit A un ensemble, $(A', \leq)$ un treillis complet.*
Soit $\sqsubseteq$ la relation définie sur l'ensemble des fonctions $A \to A'$ comme suit :
$$\forall f, g \in (A \to A') : \{f \sqsubseteq g \Leftrightarrow \forall a \in A : f(a) \leq g(a)\},$$
alors $(A \to A', \sqsubseteq)$ est un treillis complet.

Démonstration :

- On démontre facilement que $\sqsubseteq$ est une relation d'ordre.
- Soit $S \subseteq (A \to A')$ un ensemble de fonctions.
  Comme A' est un treillis complet, on peut définir la fonction suivante :
  $$h : A \to A' : x \mapsto \sqcup\{f(x) \mid f \in S\}.$$
  On a que $h = \sqcup S$, en effet :
  - h est une borne supérieure de S puisque
    $$\forall x \in A : h(x) = \sqcup\{f(x) \mid f \in S\},$$
    et donc
    $$\forall x \in A, f \in S : f(x) \leq h(x),$$
    d'où :
    $$\forall f \in S : f \sqsubseteq h.$$

- Soit h' une autre borne supérieure de S, alors par définition de la borne supérieure :

  $$\forall f \in S : f \sqsubseteq h',$$

  de sorte que l'on a également que :

  $$\forall f \in S, x \in A : f(x) \leq h'(x)$$

  et donc :

  $$\forall x \in A : h(x) = \sqcup\{f(x) \mid f \in S\} \leq h'(x),$$

  d'où :

  $$h \sqsubseteq h'.$$

3.3 Le plus petit point fixe d'un opérateur

Définition 24 *Soit $(E, \leq)$ un treillis complet.*
Un opérateur $T : E \to T$, est ***monotone*** *si*

$$\forall x, y \in E : x \leq y \Rightarrow T(x) \leq T(y)$$

Définition 25 *Soit $(E, \leq)$ un treillis complet.*
Une ***chaîne croissante*** *de E est une séquence $X_1, X_2, \ldots$, d'éléments de E, tels que : $X_1 \leq X_2 \leq \ldots$.*

Définition 26 *Soit $(E, \leq)$ un treillis complet.*
Un opérateur $T : E \to T$, est ***continu*** *si pour chaque chaîne croissante $X_1, X_2, \ldots$ de E :*

$$T(\sqcup_i X_i) = \sqcup_i T(X_i).$$

Proposition 4 *Soit $(E, \leq)$ un treillis complet, $T : E \to E$ un opérateur continu, alors T est monotone.*

Démonstration : Soit $x \leq y$.
La chaîne $x, y, y, y, \ldots$ est croissante et par conséquent :

$$\begin{aligned} T(y) &= T(\sqcup\{x, y, y, \ldots\}) \\ &= \sqcup\{T(x), T(y), T(y), \ldots\} \\ &= \sqcup\{T(x), T(y)\}, \end{aligned}$$

d'où l'on déduit que

$$T(x) \leq T(y).$$

Définition 27 *Soit E un ensemble, $T : E \to E$ un opérateur sur E.*
Un élément $a \in E$ est un ***point fixe*** *de T si $T(a) = a$*

Définition 28 *Soit $(E, \leq)$ un ensemble ordonné, $T : E \to E$ un opérateur sur E.*
Un élément $a \in E$ est le ***plus petit point fixe*** *de T si a est un point fixe de T et si pour chaque point fixe a' de T on a que $a \leq a'$.*

A nouveau on voit aisément que si le plus petit point fixe d'un opérateur existe, il est unique, nous le noterons comme $\mu\, T$.

Le théorème suivant, dû à Tarski, est d'une extrême importance : non seulement donne-t-il une condition sous laquelle le plus petit point fixe d'un opérateur existe, mais en plus il propose une méthode qui permet de l'approximer.

Théorème 2 *Soit $(E, \leq)$ est un treillis complet, $T : E \to E$ un opérateur continu, alors le plus petit point fixe de T existe.*
Ce plus petit point fixe est la borne supérieure de la séquence croissante

$$X_0 \leq X_1 \leq X_2 \leq \ldots,$$

définie comme suit :

$$\begin{cases} X_0 = \bot \\ \forall i \geq 0 : X_{i+1} = T(X_i). \end{cases}$$

Démonstration :

Considérons la séquence

$$X_0, X_1, \ldots$$

de l'énoncé.

Cette séquence est effectivement croissante :

- par la définition de $\bot$ on a que $X_0 = \bot \leq T(\bot) = X_1$
- et parce que T est monotone, on peut déduire de $X_i \leq X_{i+1}$ que

$$X_{i+2} = T(X_{i+1}) \geq T(X_i) = X_{i+1}.$$

Soit a la borne supérieure de cette séquence, alors par la continuité de T :

$$T(a) = T(\sqcup_i X_i) = \sqcup_i T(X_i) = \sqcup_i X_{i+1} = a,$$

de sorte que a est un point fixe de T.

Supposons maintenant que a' soit un autre point fixe de T.
Par la définition de $\bot$ on sait que :

$$X_0 = \bot \leq a'$$

et comme T est monotone, on peut déduire de $X_i \leq a'$ que

$$X_{i+1} = T(X_i) \leq T(a') = a'.$$

Donc :

$$\forall i : X_i \leq a'$$

et par conséquent :

$$a = \sqcup_i X_i \leq a'.$$

Ce qui termine la démonstration.

4 Un algorithme de calcul de point fixe

4.1 Introduction

Au chapitre précédent nous avons introduit les notions et les propriétés mathématiques relatives à la solution d'un système d'équations.
Notamment, nous avons vu que sous certaines conditions ce problème peut être réduit à la recherche du point fixe d'un opérateur continu.

Dans ce chapitre-ci nous allons proposer un algorithme efficace qui permet de calculer ce point fixe.
Cet algorithme générique a été développé dans [1] et prouvé sa valeur dans plusieurs applications comme [3] et [2].

4.2 Enoncé du problème

Soit A un ensemble quelconque et B un treillis complet.
Comme on l'a vu dans la proposition (3) au chapitre précédent, l'ensemble des fonctions $A \to B$ peut être muni d'un ordre qui en fait un treillis complet.

Soit τ un opérateur continu :

$$\tau : (A \to B) \to (A \to B)$$

On a vu que τ admet un plus petit point fixe, notons le par

$$\mu\ \tau : (A \to B).$$

Notre problème consiste à calculer pour un $a \in A$ donné la valeur

$$(\mu\ \tau)\ a \in B$$

que prend le point fixe en a.
Afin de réaliser ceci, nous supposons que nous disposons d'une fonction effective

$$\text{function tau } (f : A \to B, a : A) : B$$

c'est-à-dire d'un algorithme ou d'une fonction implémentée qui réalise la propriété suivante :

$$\forall f \in (A \to B),\ a \in A : \text{tau}(f, a) = (\tau\ f)(a).$$

4.3 L'approche simpliste

Selon le théorème (2), le point fixe $(\mu\tau)(a)$ recherché peut, au moins en théorie, être calculé comme suit :

Soit f_0 la fonction suivante :

$$f_0 : A \to B : a \mapsto \bot.$$

Soit pour $i \in \mathbb{N}_0$: $f_{i+1} = \tau(f_i)$, alors

$$\mu\ \tau = \sup_i f_i$$

et par conséquent :

$$(\mu\ \tau)\ a = \sup_i f_i(a).$$

En forme algorithmique ceci deviendrait :

```
function CalculerPointFixe(a :A) :B
begin
    ∀b ∈ A : f(b) ← ⊥
    repeat
        f0 ← f
        ∀b ∈ A : f(b) ← tau(f0, b)
    until f = f0
    return f(a);
end
```

Cette esquisse d'algorithme prend donc à chaque itération l'approximation f_0 du point fixe $\mu\ \tau$ trouvée jusque là et l'utilise pour calculer f, l'approximation suivante.
Bien que d'une simplicité émouvante, il est clair que cette façon de procéder est d'une inefficacité absolument inacceptable et ceci pour deux raisons :

- D'abord on peut remarquer que f est calculée pour tous les éléments de A, même pour ceux qui n'ont aucune importance pour la valeur que l'on veut calculer.
  Considérons à titre d'illustration la situation suivante :
  L'ensemble A est composé de n éléments : $A = \{a_1, a_2, \ldots, a_n\}$
  tau(f, a_1) et tau(f, a_2) ne dépendent que des valeurs de f en a_1 et a_2, ou plus formellement, il existe des fonctions g_1 et g_2 telles que :
  $$\begin{cases} \text{tau}(f, a_1) = g_1(f(a_1), f(a_2)) \\ \text{tau}(f, a_2) = g_2(f(a_1), f(a_2)) \end{cases}$$
  de plus, il existe des expressions pour tau(f, a_i) où $i > 2$.
  Si l'on s'intéresse maintenant à connaître la valeur de $(\mu\ \tau)(a_1)$, il est clair qu'il suffit de considérer seules les deux premières équations et de calculer les valeurs du point fixe en a_1 et a_2.
  L'algorithme ci-dessus par contre considère toutes les équations et ne s'arrêtera que quand le point fixe sera connu dans tous les éléments de A, ce qui est extrêmement inefficace si A est composé de beaucoup d'éléments et simplement impossible si A est infini.
- Ensuite on remarque que, à chaque itération, $f(b)$ est calculé pour chaque b, même pour ceux pour lesquels la valeur de f ne peut avoir changée.
  Illustrons ceci à nouveau sur l'exemple exposé ci-dessus.
  Supposons que, dans une itération de l'algorithme, seule la valeur de $f(a_3)$ ait changé, les autres valeurs $f(a_i)$ restant les mêmes.
  Alors il n'y a, dans la prochaine itération, aucun besoin de recalculer $f(a_1)$ ou $f(a_2)$, puisqu'on sait d'avance que ces valeurs n'auront pas changé.

4.4 L'algorithme de calcul de point fixe

Pour parer aux inefficacités d'une approche trop simpliste comme ci-dessus, nous avons utilisé dans notre programme un algorithme de point fixe bien plus sophistiqué. L'algorithme en question est repris de l'article [1], et sera brièvement exposé dans cette section.

4.4.1 Présentation globale

On considère une variable globale

$$\text{Pt} : A \to B$$

qui représente (qu'il soit comme table, ensemble, ...), l'approximation de $\mu(\tau)$ que l'on a déjà trouvé. Cette fonction est initialisée comme

$$\text{Pt} : A \to B : a \mapsto \bot$$

et sera améliorée au cours du programme jusqu'à ce qu'on soit sûr que

$$\text{Pt}(\alpha) = (\mu(\tau))(\alpha),$$

où α est la valeur dans laquelle on veut connaître le point fixe.

L'algorithme s'articule sur deux fonctions :

```
function tau(a :A) :B
procedure Calculer(a :A)
```

qui s'appellent mutuellement.

La fonction *tau* a, à part le fait que l'on ne spécifie plus explicitement le paramètre fonctionnel, la même signification qu'au paragraphe précédent. Elle implémente l'opérateur τ sur la fonction Pt, c'est-à-dire :

$$\forall a \in A : \text{tau}(a) = (\tau(Pt))(a)$$

Elle permet ainsi de trouver à partir de l'approximation *Pt* du point fixe, une approximation suivante (qui est soit la même, soit meilleure) en prenant *tau(a)* comme nouvelle valeur en a.
L'algorithme suppose maintenant que chaque fois que cette fonction *tau* a besoin d'une valeur *Pt(b)*, elle fait d'abord un appel *Calculer(b)*.

La procédure *Calculer(a)* constitue la charnière de l'algorithme, elle a comme but de calculer de nouvelles approximations pour *Pt(a)* tant que ceci est possible.
Concrètement, elle appelle la fonction *tau(a)*, tant que

- une autre exécution de *tau(a)* n'est pas active (ceci pour éviter qu'on entre dans une boucle)
- elle ne sait pas d'avance si le résultat obtenu sera identique à la valeur de *Pt(a)* dont on dispose déjà.

La fonction *Calculer* doit donc éviter que deux instances de la fonction *tau(a)* pour le même paramètre *a* soient actives au même moment et doit disposer d'un moyen pour vérifier si la valeur de *Pt(a)* est susceptible d'être améliorée par un nouvel appel à *tau(a)*.
Pour réaliser ceci, elle doit disposer d'une structure de données qui reprend les calculs exécutés et en cours d'exécution ainsi que les dépendances entre eux.
La structure adaptée à ceci est un graphe orienté dont les sommets sont étiquetés par des éléments de A.
La signification précise de ce graphe est la suivante :

- Un sommet *a* correspond à la dernière exécution d'un appel *tau(a)*.
- Un arc d'un sommet *a* à un sommet *b*, signifie que la dernière exécution de *tau(a)* a utilisé la valeur de *Pt(b)*.
- On ne garde dans le graphe que les sommets *a* dont on sait que la valeur de *Pt(a)* ne peut plus être améliorée en ce moment.
  - Soit parce que l'exécution *tau(a)* est toujours active, ce qui fait qu'on ne peut pas en lancer une deuxième sans risquer de boucler.
  - Soit parce que, bien que l'exécution *tau(a)* soit terminée, aucune des valeurs *Pt(a')* qu'elle a utilisée n'a changé depuis son utilisation et aucune de ces valeurs ne peut être améliorée en ce moment.

En plus de ce graphe,on utilise encore une variable globale *SommetAppelant* qui contient le sommet *a* tel que le dernier appel fait à la fonction *tau* est *tau(a)*.

Pour ce qui est de l'implémentation de l'arbre, nous utilisons le type suivant :

```
TSommet = CP[
            Elem : A
            Utilise, UtilisePar : SET[TSommet]
            ]
```

qui représente un sommet du graphe, étiqueté par un élément de *A*, avec un ensemble d'arcs sortants (dont les destinations sont reprises dans l'ensemble *Utilise*) et un ensemble d'arcs entrants (dont les origines sont reprises dans l'ensemble *UtilisePar*).
Avec ce type, le graphe que l'on appellera graphe des dépendances *Dg*, devient un ensemble de sommets :

```
Dg : SET[TSommet]
```

La variable *SommetAppelant* est simplement un sommet :

```
SommetAppelant : TSommet
```

4.4.2 Présentation détaillée

Nous pouvons maintenant passer aux détails de l'algorithme :

```
Pt : A → B
Dg : SET[TSommet]
SommetAppelant : TSommet
```

```
function CalculerPointFixe(a :A) :B ;
begin
    Dg := [ ];
    SommetAppelant := Null;
    for a in A do Pt(a) ←⊥ ;
    Calculer(a) ;
    return Pt(a) ;
end ;

procedure Calculer(a :A) ;
begin
    for S in Dg do
        if (S.Elem = a)
        then begin
                SommetAppelant.Utilise +=S ;
                S.UtilisePar +=SommetAppelant ;
                return ;
             end ;
    AncSommetAppelant ←SommetAppelant ;
    repeat
        S ←CreateSommet(a) ;
        Dg +=S ;
        SommetAppelant ←-S ;
        b ←-tau(a) ;
        if (b ≠ Pt(a)) ∧ (S.UtilisePar ≠ [ ])
        then begin
                Pt(a) ←b ;
                Remove(S) ;
             end
        else  begin
                Pt(a) ←b ;
                break ;
             end ;
    end-repeat ;
    SommetAppelant ←AncSommetAppelant ;
    if (SommetAppelant ≠ NULL) then
        begin
            SommetAppelant.Utilise +=S ;
            S.UtilisePar +=SommetAppelant ;
        end ;
end ;

procedure Remove(S :TSommet) ;
begin
    for S' in S.Utilise
        do S'.UtilisePar −=S ;
    while S.UtilisePar ≠ [ ]
        do Remove(S.UtilisePar[0]) ;
    Dg −=S ;
end ;

function CreateSommet(a :A) :TSommet ;
begin
    S.Elem ←-a ;
    S.Utilise ←-[ ] ;
    S.UtilisePar ←-[ ] ;
    return S ;
end ;
```

Il y a quatre fonctions à considérer :

La fonction ***CreateSommet(a)*** ne fait que créer un nouveau sommet.

La procédure ***Remove(a)*** enlève le sommet a et tous ses ancêtres du graphe.
On peut comprendre sa pertinence pour l'algorithme :

Si on vient de modifier la valeur de *Pt(a)* ou d'apprendre (qu'il n'est pas exclus) que cette valeur peut être améliorée, alors pour chaque sommet b dont le dernier appel à *tau(b)* a directement ou indirectement utilisé *Pt(a)*, la valeur de *Pt(b)* est aussi susceptible d'être améliorée et on doit supprimer b du graphe pour indiquer ceci.

La fonction ***CalculerPointFixe(a)*** initialise le graphe et la fonction *Pt* et fait un appel *Calculer(a)*, une fois celui-ci a terminé, la valeur $(\mu\ \tau)(a)$ recherchée se trouve dans *Pt(a)*.

La fonction ***Calculer(a)*** fait bien évidemment la plupart du travail. On procède de la façon suivante : D'abord on regarde si le sommet a se trouve dans le graphe.

- Si ceci est le cas,
  - soit il y a déjà une exécution de *tau(a)* qui est active et on ne peut pas en lancer une autre sans risquer de boucler,
  - soit la dernière exécution est terminée, mais on sait que la valeur qu'elle a renvoyée ne peut être améliorée par un nouvel appel.

  Dans les deux cas, la procédure peut se terminer immédiatement, après avoir indiqué sur le graphe que le sommet appelant (qui correspond à l'exécution *tau(b)* qui a lancé la procedure), va utiliser la valeur de *Pt(a)*.
- Si ce n'est pas le cas, on met le sommet a dans le graphe pour indiquer qu'une exécution de *tau(a)* sera active et on fait l'appel.
  Son résultat, mis dans *Pt(a)*, nous donne alors la nouvelle approximation.
  Ensuite, il y a deux cas à considérer :
  - Soit on vient de trouver une nouvelle valeur pour *Pt(a)* tandis que le graphe nous indique que l'ancienne valeur a été utilisée dans le calcul d'autres valeurs. Dans ce cas là, on doit enlever tous les ancêtres de a, c'est-à-dire tous les éléments dont la valeur dépend directement ou indirectement de cette ancienne valeur, du graphe. En plus, comme le sommet qui a utilisé *Pt(a)* ne peut être qu'un descendant de a, on aura ainsi supprimé a lui-même du graphe.
    On se trouve dans la situation où la nouvelle valeur de *Pt(a)* a été calculée sur base de l'ancienne. Comme on vient de découvrir que cette ancienne valeur était fausse, on ne peut pas être sûr d'avoir trouvé la meilleure approximation de *Pt(a)* possible et on doit recommencer.
  - Dans l'autre cas, la valeur obtenue est effectivement la meilleure que l'on puisse trouver pour l'instant et il ne reste plus qu'à indiquer sur le graphe que l'exécution de *tau(b)* à laquelle nous allons repasser la main est sur le point d'utiliser la valeur obtenue.

4.4.3 Exemple d'exécution

On termine l'explication de cet algorithme avec un petit exemple de son exécution.

Soit A un ensemble à deux éléments :

$$A = \{a_1, a_2\}$$

On considère le treillis complet des fonctions $A \to \mathcal{P}(\{0,1\})$. Et on s'intéresse plus particulièrement à la solution du système d'équations suivant :

$$\begin{cases} f(a_1) &= f(a_2) \cap \{0\} \\ f(a_2) &= f(a_1) \cap \{1\} \end{cases}$$

Ou, plus précisément au point fixe de l'opérateur

$$\tau : (A \to \mathcal{P}(\{0,1\})) \to (A \to \mathcal{P}(\{0,1\}))$$

défini par :

$$\begin{cases} (\tau(f))(a_1) &= f(a_2) \cap \{0\} \\ (\tau(f))(a_2) &= f(a_1) \cap \{1\} \end{cases}$$

La fonction effective *tau*, peut être définie comme suit :

```
function tau(a :A) : P({0,1})
begin
    if a = a1
    then begin
            Calculer(a2) ;
            return Pt(a2) ∪{0} ;
         end
    else  begin
            Calculer(a1) ;
            return Pt(a1) ∪{1} ;;
         end ;
end ;
```

Si on trace l'exécution de l'algorithme sur cet exemple, après un appel *CalculerPointFixe*(a_1), on obtient le tableau suivant :

		Dg	Pt(a_1)	Pt(a_2)
1 :	Calculer(a_1)			
2 :	.....	a_1	$\emptyset$	$\emptyset$
3 :	tau(a_1)			
4 :	Calculer(a_2)			
5 :	.....	a_1 a_2	$\emptyset$	$\emptyset$
6 :	tau(a_2)			
7 :	Calculer(a_1)			
8 :	.....	$a_1 \longleftarrow a_2$	$\emptyset$	$\emptyset$
9 :	return {1}			
10 :	.....	$a_1 \longleftrightarrow a_2$	$\emptyset$	{1}
11 :	return {0,1}			
12 :	.....	a_1	{0, 1}	{1}
13 :	tau(a_1)			
14 :	Calculer(a_2)			
15 :	.....	a_1 a_2	{0, 1}	{1}
16 :	tau(a_2)			
17 :	Calculer(a_1)			
18 :	.....	$a_1 \longleftarrow a_2$	{0, 1}	{1}
19 :	return {0,1}			
20 :	.....	$a_1 \longleftrightarrow a_2$	{0, 1}	{0, 1}
21 :	return {0,1}			
22 :	.....	$a_1 \longleftrightarrow a_2$	{0, 1}	{0, 1}

- La fonction *CalculerPointFixe*(a_1) va, après avoir initialisé *Dg* et *Pt*, appeler *Calculer*(a_1) (1).
- Cette fonction va mettre le sommet a_1 sur le graphe (2) et faire un appel *tau*(a_1) (3).
- Comme on le voit dans le code pour la fonction *tau*, la première chose qu'elle va faire, est d'appeler *Calculer*(a_2) (4).
- Cette fonction va mettre a_2 sur le graphe (5) et faire un appel à *tau*(a_2) (6), qui va à son tour appeler *Calculer*(a_1) (7).
- Maintenant la fonction *Calculer*(a_1) va se rendre compte que le sommet a_1 est déjà sur le graphe (ce qui veut dire dans ce cas-ci, qu'une exécution de *tau*(a_1) est déjà en cours) et elle va s'arrêter après avoir indiqué sur le graphe que la fonction *tau*(a_2) qui l'a appelée, va utiliser la valeur $Pt(a_1) = \emptyset$ (8).
- La fonction *tau*(a_2) reprend la main et va exécuter l'instruction *return* $Pt(a_1) \cup \{1\}$ (9).
- Quand la fonction *Calculer*(a_2) reçoit cette valeur {1} elle va mettre à jour la valeur *Pt*(a_2) (10) Comme le sommet a_2 n'est utilisé par aucun sommet, cette mise à jour ne met en cause aucun résultat trouvé, et elle peut s'arrêter après avoir indiqué sur le graphe que la fonction *tau*(a_1) qui l'a appelée, va utiliser le résultat *Pt*(a_2). On obtient ainsi le graphe des dépendances en (10).
- La fonction *tau*(a_1), reprend la main et va exécuter l'instruction *return* $Pt(a_2) \cup \{0\}$, pour renvoyer la valeur {0, 1} (11).
- La fonction *Calculer*(a_1) va utiliser cette valeur pour mettre à jour la valeur *Pt*(a_1)(12).
  Comme elle voit sur le graphe que l'ancienne valeur de *Pt*(a_1) a été utilisée pour calculer la valeur

de $Pt(a_2)$ et que cette dernière valeur à servi au calcul de $Pt(a_1)$, elle déduit que l'on ne peut plus être sûr d'aucun résultat obtenu.
Elle supprime les deux sommets du graphe et fait une deuxième itération qui commence par la mise sur le graphe du sommet a_1 et un appel $tau(a_1)(13)$.

Les détails concernant cette nouvelle itération sont repris dans la figure ci-dessus.

4.4.4 Rapport avec l'algorithme original

La version originale de l'algorithme que nous venons d'expliquer se trouve dans [1]. Bien que complètement équivalente sur le fond, la présentation que nous en avons donnée est assez différente sur la forme. Quelques mots d'explication semblent appropriés :

- D'abord on remarquera que nous avons réuni les deux fonctions *pretended_f* et *repeat_computation* au sein d'une même fonction *Calculer*, ceci a été fait pour des raison d'efficacité : en implémentant l'algorithme comme ceci, le nombre d'appels de fonction se voit réduit d'une façon non-négligable.
- Deuxièmement, l'algorithme original contient des optimisations pour le cas où l'ensemble A est muni d'un ordre partiel et on se limite à considérer des fonctions $A \rightarrow B$ qui sont monotones. Ceci complique assez fort la situation par rapport au cas de base que nous considérons. Dans notre algorithme, quand dans la procédure *Calculer(a)*, la valeur de *Pt(a)* change, tous les sommets du graphe qui se trouvent sur un circuit avec le sommet a en seront enlevés. Dans l'algorithme original, en plus de ces sommets d'autres peuvent aussi être enlevés, il s'agît de sommets qui n'ont même aucun lien direct avec le sommet en question, c'est-à-dire qui n'en sont ni des ascendants, ni des successeurs.
  Une des conséquences de ceci est que si l'on se limite au cas de base, l'invariant $\alpha \in \text{ics} \Rightarrow \alpha \in \text{dom(Dg)}$ sera toujours respecté au début de la procédure *repeat_computation*, ce qui fait que la variable *ics* n'as plus de raison d'être.
- Une troisième différence entre les deux versions est que nous avons utilisé une variable globale *SommetAppelant* au lieu de s'arranger pour que la fonction *tau* passe le paramètre pour lequel elle a été appelée, à la procédure *Calculer* (cfr. le paramètre α dans *pretended_f*). La raison de ceci est que notre fonction *tau* est assez complexe (elle est en effet composée de trois fonctions) et que dans notre cas les éléments de A sont composés de deux (ou trois) parties. Il nous a semblé un peu plus clair d'utiliser une variable globale au lieu d'encombrer nos fonctions de paramètres supplémentaires qui doivent être passés de fonction en fonction.
- Les autres différences sont mineures :
  - Les fonctions *Adjust_pt* et *Add_dg* sont mises *inline*.
  - Notre version ne spécifie plus comment la fonction *Pt* doit être implémentée, en effet, dans notre programme les valeurs *Pt(a)* ne se trouveront pas toutes prêtes dans une table mais devront être calculées à partir des valeurs dans une autre table.
  - Un sommet prend aussi, à part son étiquette et l'ensemble de ses suivants, l'ensemble de ses précedents, ceci parce que l'on utilise cet ensemble dans la fonction *Remove*.
  - La fonction *Remove* supprime non seulement les successeurs d'un sommet, mais aussi le sommet lui-même. Comme on va toujours l'appeler pour un sommet qui se trouve sur un circuit, les deux approches sont équivalentes quant à l'utilisation, mais notre approche semble avoir une implémentation un peu plus efficace.

Première partie
Le β-langage

5 La syntaxe concrète

5.1 Introduction

La syntaxe concrète est celle qu'il faut absolument respecter lors de la rédaction de code sur machine. Sans le respect de cette syntaxe, l'utilisateur est immédiatement sanctionné par un message d'erreur. Cette syntaxe est aussi le pilier de notre programmation pour ce qui concerne l'outil de parsage. Ce qui est dit ici se retrouve dans les outils de programmation LEX et YACC

5.2 La syntaxe concrète

Conventions Dans cette syntaxe, on utilise les conventions suivantes :

< entier >	::=	$0 \mid \{1..9\}\{0..9\}^*$
< idmin >	::=	$\{a..z\}\{a..zA..Z0..9\_\}^*$
< idmaj >	::=	$\{A..Z\}\{a..zA..Z0..9\_\}^*[']^*$
< nom_prédicat >	::=	< idmin > | < idmaj >
< nom_domaine >	::=	< idmin > | < idmaj >
< nom_intervalle >	::=	< idmin >
< nom_constante >	::=	< idmin >
< nom_variable >	::=	< idmaj >

Ces conventions sont présentées dans un premier temps afin de mettre en exergue les petits détails comme

- comment écrire un nom de variable ;
- quelles sont les différentes familles d'identificateurs.

Ainsi un prédicat ne pose aucune restriction, tandis qu'une distinction est imposée entre constantes et variables :

$c, a, toto, aLBERT$	sont des constantes
$X, Y, NABUCHODONOSOR, B212$	sont des variables

Déclarations Maintenant que cette question est règlée, nous pouvons entrer au cœur du problème. Sur cette base, nous obtenons la syntaxe concrète dont nous allons parler morceau par morceau. Cela permattra de découvrir petit à petit la syntaxe, mais aussi de préciser au moment opportun les choses qui n'y apparaissent pas.

$$\begin{cases} < \text{programme} > & ::= \{< \text{définition} >\}^* \\ < \text{définition} > & ::= < \text{défintion\_domaine} > \mid < \text{défintion\_prédicat} > \end{cases}$$

Le programme n'est rien d'autre qu'une suite de déclarations de domaines ou de prédicats. Le programmeur n'est pas obligé de déclarer tous les domaines d'abord, séparement des prédicats logiques. Il peut les mélanger comme bon lui semble pourvu qu'il respecte la règle bien connue : "*Déclarer avant d'utiliser*". Une seule exception est tolérée : celle où deux prédicats s'appellent l'un l'autre. Ainsi un message d'erreur sera envoyé quand un domaine sera utilisé sans avoir encore été déclaré auparavant (voir erreur 13 page 43), ou à la fin de l'examen du code si un prédicat est appelé sans avoir été déclaré nulle part (voir erreur 15 page 43). Nous allons exposer ce qu'il en est des domaines, ensuite nous aborderons les prédicats.

Domaines

< définition_domaine >	::=	< domaine_par_défaut >
	|	< domaine_nommé >
< domaine_par_défaut >	::=	**domain** < description_domaine > ;
< domaine_nommé >	::=	**domain** < nom_de_domaine > < description_domaine > ;
< description_domaine >	::=	< description_enumérée >
	|	< description_intervalle >
< description_enumérée >	::=	{ < nom_constante > {, < nom_constante >}* }
< description_intervalle >	::=	< nom_intervalle > [< entier > .. < entier >]

L'utilisateur a la possibilité de définir un domaine par défaut. C'est celui dont héritera toute variable de prédicat pour laquelle le type (domaine) ne sera pas précisé. Il est conseillé de le définir au début du

programme afin que la première déclaration de prédicat, si elle comporte des typages implicites, puisse bénéficier de la déclaration de ce domaine par défaut.
Dans tous les cas, il faut utiliser le mot réservé *domain*. On remarquera la particularité des intervalles car ils bénéficient de deux noms :

1. un nom de domaine ;
2. un nom d'intervalle.

Ceci permet une reconnaissance plus facile par l'analyseur, mais aussi une lecture plus facile du code car en un coup d'œuil on distingue le nom de l'intervalle. Voici quelques exemples :

domain $\{yin, yang\}$
domain $ALPHA\ a[2..15]$
domain $je\_vous\_dis\_que\_ceci\_est\_mon\_domaine\{maison, jardin\}$
domain $Belgique\ \{flandre, wallonie, bruxelles\}$
domain $King\{leopold\_I, leopold\_II, albert\_I, leopold\_III, baudouin\_I, albert\_II\}$

On refusera

domain $\{Joe, Jack, William, Averell\}$
domain $fourchette\ F[2..13]$
domain $Fortran\ [77..90]$

Prédicats

< définition_prédicat >
::= < nom_prédicat > (< liste_parametres_typés >) ::= < disjonction > ;
< liste_parametres_typés > ::= ε
| < param_typé > {, < param_typé >}*
< param_typé > ::= < nom_de_variable >
| < nom_de_variable > : < nom_de_domaine >
< disjonction > ::= < conjonction > {| < conjonction >}*
< conjonction > ::= < litéral >
| < atome >
| < conjonction > {& < conjonction >}*

Cette partie de la grammaire montre bien quels sont les choix posés quant à la façon d'exprimer un prédicat. Tous les prédicats déclarés doivent être présentés sous forme disjonctive. On comprendra la nécessité de procéder classiquement avec papier et crayon avant l'encodage du programme sur support éléctronique. De plus, on fera un parallèle avex le langage PROLOG où l'énonciation de plusieurs clauses de même nom équivaut à l'expression d'un seul et même prédicat énoncé sous forme disjonctive. Toutefois ici, l'arité d'un prédicat ne le distingue pas des autres prédicats de même nom et d'arité différente comme c'est le cas de PROLOG.
Les arguments doivent être typés, c'est-à-dire suivis du nom du domaine auquel ils appartiennent. Si cette précision est omise, alors ils héritent du domaine par défaut si ce dernier est déclaré. Voici quelques exemples :

$p(X : ALPHA,\ REGION : Belgique,\ Z,\ T : Belgique) := [...]$
$b212(A, B, C, D, E, F, G, H, I, J, K, L, M) := [...]$
$Grand\_Nom(N : King) := [...]$

A présent, nous allons parler de ce que contiennent les prédicats.

Le corps des prédicats

< litéral > ::= < appel_prédicat >
| not < appel_prédicat >
| ~ < appel_prédicat >
< appel_prédicat > ::= < nom_prédicat > (< liste_paramètres_effectifs >)
< liste_paramètres_effectifs > ::= ε
| < var_ou_const > {, < var_ou_const >}*
< atome > ::= < nom_variable > in < description_domaine >
| < nom_variable >< operateur >< var_ou_const >

On voit ici les différents objets qui existent dans la déclaration d'un prédicat.

1. Soit on procède à un appel récursif avec une liste de paramètres effectifs. Si le nombre de paramètres effectifs n'est pas le même que lors de la déclaration du prédicat appelé, l'utilisateur est sanctionné par un message d'erreur.
2. Soit on énonce un atome, c'est-à-dire une opération d'appartenance à un domaine ou une comparaison entre variables.

Enfin, les derniers éléments intervenant sont :

< var_ou_const >	::=	< nom_variable > | < constante >
< constante >	::=	< const_elem > | < constante_indexée >
< const_elem >	::=	< nom_constante >
< constante_indexée >	::=	< nom_intervalle > [< entier >]
< opérateur >	::=	= | <>

Lors de l'utilisation de ces règles, il faut évidemment respecter quelques sous-entendus, à savoir

- le nom d'une constante dans un appel de prédicat n'a de sens que si elle appartient à un domaine préalablement défini ;
- l'appartenance d'une variable à un intervalle ou à un sous-domaine n'a de sens que si ledit intervalle ou domaine est préalablement déclaré dans le code.

Une déclaration type de prédicats est

$$p(X, Y : mon\_domaine) := X = a \mid X = b \mid X = c \;\&\; Y \; in \; inter[7..9]$$
$$q(A : mon\_domaine, B) := B <> c \mid not \; p(Z, A) \;\&\; q(Z, a)$$

Les commentaires Lorsque le programmeur désire placer certaines lignes en commentaire, il peut le faire à l'aide des symboles /* (ouverture) et */ (fermeture) ou à l'aide du symbole // (reste de la ligne).

5.3 Les règles cachées de la syntaxe

En énonçant goutte à goutte la syntaxe concrète, nous avons mis au clair la structure d'un programme, mais aussi quelques conventions implicites non énonçables par le biais de la grammaire. Il reste quelques points à préciser.
D'abord, tous les noms de domaines doivent êtres distincts entre eux, qu'ils dénomment des ensembles énumérés ou des intervalles (voir erreur 3). De plus, il ne peut exister qu'une seule déclaration du domaine par défaut(voir erreur 4).
Similairement, tous les noms de prédicats existants doivent être distincts entre eux (voir erreur 5). Il n'existe pas de prédicat bénéficiant d'un statut supérieur aux autres.
Les nom des constantes énumérées doivent être distincts entre eux (erreur 6). De même pour les noms des intervalles. Ceci permet d'éviter toute ambiguïté lors d'une clause avec les opéateurs =, <> : on détermine immédiatement à quel domaine ou intervalle appartient la constante citée.
Lors de la déclaration d'un prédicat, toutes les variables de l'en-tête doivent être distinctes (erreur 7). On n'admettra dès lors, aucune d'éclaration de la forme

$$\text{mon\_prédicat}(X : \text{mon\_dom}, Y, X : \text{mon\_dom}) := \ldots$$

où Y appartient au domaine par défaut mais où la variable X est utilisée deux fois.
Au niveau des constantes, il est bien sûr plus sensé d'utiliser des constantes appartenant au domaine adéquat. Autrement dit, il serait idiot d'écrire $X = c$ alors que c n'appartient pas au domaine de X (erreur 9). De plus, dans le cas d'une appartenace à un intervalle, les bornes doivent respecter la déclaration située plus tôt dans le code tout en évitant de constituer un sous-intervalle vide (erreur 10). Avec ***domain*** mon_domaine inter[4..25], on admettra $X =$ inter[6] ainsi que Y ***in*** inter[7..18], tandis que l'on refusera $X =$ inter[2], Y ***in*** inter[3..29] ou encore Y ***in*** inter[17..10].

5.4 Gestion des erreurs

Lorsque l'analyseur syntaxique rencontre des erreurs, ils s'arrête *ipso facto*. Il signale le numéro de la ligne où l'erreur s'est produite, donne l'explication de l'erreur, puis écrit la ligne entière où l'erreur s'est produite. Les différents codes d'erreur sont les suivants :

Erreur 3

Les noms des domaines dans les définitions des domaines doivent être distincts.
—→ **nom de domaine deja utilise.**
Exemple : [−3−] Ligne 4 : nom de domaine deja utilise (patati).
domain patati {gris, jaune, pale} ;

Erreur 4

Le domaine par défaut ne peut être défini qu'une et une seule fois.
—→ **domaine par defaut defini une seconde fois.**
Exemple : [−4−] Ligne 3 : domaine par defaut defini une seconde fois.
domain {noir, jaune, rouge} ;

Erreur 5

Les noms des prédicats doivent tous être distincts.
—→ **nom de predicat deja utilise.**
Exemple : [−5−] Ligne 6 : nom de predicat deja utilise (p).
p(X,Y) := X=Y ;

Erreur 6

Un nom de constante ou d'intervalle ne peut se répéter plusieurs fois dans les déclaration.
—→ **nom de constante ou d'intervalle deja utilise.**
Exemple : [−6−] Ligne 3 : nom de constante ou d'intervalle deja utilise (a).
domain albert {a,p,q,r,a[6]

Erreur 7

Les noms de variables dans l'en-tête d'un prédicat doivent être distinctes.
—→ **meme nom de variable plusieurs fois dans les parametres.**
Exemple : [−7−] Ligne 5 : meme nom de variable plusieurs fois dans les parametres (X).
q(X,Y,X,[7]

Erreur 8

Une constante utilisée doit avoir été déclarée auparavant dans un domaine.
—→ **constante non declaree.**
Exemple : [−8−] Ligne 4 : constante non declaree (p[349]).
bibi(K) := X=p[349]

Erreur 9

Toute énumération de constantes en rapport avec une appartenance ne peut contenir que des constantes du même domaine.
—→ **les constantes ne sont pas du même type.**
Exemple : [−9−] Ligne 5 : les constantes du domaines ne sont
pas de même type (bleu declaree ailleurs).
g(X,Y) := X in {a,b,c, bleu}

Erreur 10

Les bornes définissant un intervalle doivent avoir du sens (borne inférieure $\leq$ borne supérieure).
—→ **l'intervalle est vide.**
Exemple : [−10−] Ligne 3 : l'intervalle est vide (toto).
domain empty toto[10..6]

Erreur 11

On ne peut utiliser des constantes indexées qu'en se servant du nom de l'intervalle et en spécifiant un entier compris dans entre les deux bornes qui déclarent l'intervalle.
—→ **nom d'intervalle inexistant ou bornes incorrectes.**
Exemple : [−11−] Ligne 4 : nom d'intervalle inexistant ou bornes incorrectes (j[7..20]).
p(X) := X in j[7..20]

Erreur 12

[6] Le parseur s'arrête dès qu'il constate une erreur. Aussi ne prend-il même pas la peine de fermer l'accolade.
[7] Même remarque.

Il en va de même lors d'une appartenance à un intervalle.
—→ **intervalle non déclaré ou bornes incorrectes.**
Exemple : [-12-] Ligne 5 : nom d'intervalle deja existant (i).
domain zigomar i[4..8]

Erreur 13

Une variable ne peut être typée qu'avec un nom de domaine déjà déclaré.
—→ **domaine non declare.**
Exemple : [−13−] Ligne 5 : domaine non declare (the_domaine).
p(X : the_domaine

Erreur 14

On ne peut omettre le typage d'une variable que si un domaine par défaut à été défini plus haut dans le code.
—→ **domaine par defaut non declare.**
Exemple : [−14−] Ligne 3 : domaine par defaut non declare (X non typable).
p(X)

Erreur 15

Chaque appel de prédicat doit référer à un prédicat déclaré dans le code, éventuellement après.
—→ **predicat appele mais non declare.**
Exemple : [−15−] Ligne 3 : le predicat toto n'est pas non declare.

Erreur 16

Le nombre d'arguments d'un prédicat doit être respecté lors d'un appel.
—→ **arité du prédicat appelé non respectée.**
Exemple : [−16−] Ligne 3 : l'arité du predicat toto est 3.

Erreur 17

Une constante ne peut être utilisé comme paramètre dans un appel de prédicat que si le type de cette constante correspond au type du paramètre.
—→ **constante utilisée comme paramètre est de mauvais type.**
Exemple : [−17−] Ligne 3 : la constante c est de type mon_domaine,
mais est utilisée comme paramètre de type the_alien_domaine

5.5 Le typage des variables

Une dernière contrainte syntaxique que le programmeur devra respecter est que les variables dans le programme doivent être bien typées.
Ceci nécessite peut-être un peu plus d'explication :

Comme nous l'avons vu, l'utilisateur a la possibilité d'utiliser plusieurs domaines dans son programme. Ceci implique que notre interpréteur devra essayer de déterminer de façon univoque le type de chaque variable qui apparaît dans ce programme.

Il importe de remarquer que ceci ne veut *pas* dire qu'il va associer un type à chaque nom de variable. En effet, il y a une différence importante entre une variable et son nom, dans le sens où :

- Deux variables qui apparaissent dans des <*définition_prédicat*> différentes, sont considérées distinctes, même si elles portent le même nom.
- Deux variables qui apparaissent dans une même <*définition_prédicat*> et qui portent le même nom, sont toutefois considérées distinctes si elles apparaissent dans des <*conjonction*> différentes et si leur nom n'apparaît pas dans les <*param_typé*> de ce prédicat

Ainsi dans l'exemple

$$\begin{cases} p(X,Y) & := & X = a \ \& \ Y = b; \\ q(X) & := & p(X,Y) \mid X <> Y; \end{cases}$$

on peut distinguer cinq variables au lieu de deux comme suit :

$$\begin{cases} p(X_1,X_2) & := & X_1 = a \ \& \ X_2 = b; \\ q(X_3) & := & p(X_3,X_4) \mid X_3 <> X_5; \end{cases}$$

La raison pour laquelle les deux variables avec le nom Y dans la définition de q sont considérées distinctes est que cette définition est en fait une forme sténographique pour la formule logique :

$$q(X) \Leftrightarrow [\exists Y : p(X, Y)] \vee [\exists Y : X \neq Y].$$

Ayant fait le point sur ce sujet, nous pouvons maintenant citer les règles qui concernent le calcul du type d'une variable :

- Une variable qui apparaît comme paramètre dans la définition d'un prédicat a le type de ce paramètre.
- Une variable qui apparaît dans un atome de type

  < nom_variable > in < description_domaine >
  < nom_variable > < opérateur > < constante >

  a le types des constantes qui apparaissent dans la partie droite de cet atome.
- Les deux variables qui apparaissent dans un atome de type

  < nom_variable > < opérateur > < nom_variable >

  sont de même type.
- Une variable qui apparaît dans un

  < appel_prédicat >,

  a le type du paramètre correspondant pour le prédicat appelé.

On dira que le programme est bien typé si ces règles permettent de déterminer de façon univoque le type de chaque variable du programme.

Dans le cas contraire, deux problèmes peuvent se présenter :

- Soit il y a un conflit de type, c'est à dire, une variable à laquelle les règles citées associent deux types différents.

  Dans ce cas l'utilisateur sera sanctionné par un message d'erreur :

  Erreur 18

  ⟶ `conflit de type.`
  Ligne *** : conflit de type pour la variable *** : *** ou ***.

  Si le conflit est apparu lors de l'analyse d'un atome

  <nom_variable> <opérateur> <nom_variable>,

  le message d'erreur fournira encore plus d'information sous la forme

  ⟶ `conflit de type.`
  Ligne *** : conflit de type pour les variables *** et *** : *** ou ***.

- Soit le type de certaines variables reste inconnu après l'application des règles citées.

  Dans ce cas-ci, le message d'erreur renvoyé sera

  Erreur 19

  ⟶ `type indéterminable.`
  Ligne *** : le type pour les variables *** et *** est indéterminable.

  Ce message indique toujours un problème pour un couple de variables, on comprend en effet facilement que le type d'une variable ne peut être indéterminable que si cette variable apparaît exclusivement dans des atomes de la forme

  <nom_variable> <opérateur> <nom_variable>,

  où le type des autres variables est également indéterminable.

6 La syntaxe abstraite

6.1 Introduction

Jusqu'ici nous avons considéré la syntaxe concrète du programme, c'est à dire spécifié les contraintes syntaxiques que doit respecter le texte introduit par l'utilisateur pour être considéré comme un programme de notre langage. Ces contraintes contiennent des règles de production, l'unicité des noms, le fait que le programme doit être bien typé et le fait que l'on ne peut pas utiliser un domaine, une constante ou un predicat avant de l'avoir défini.
Maintenant que nous allons passer au niveau sémantique et plus loin au niveau calcul, nous avons intérêt à faire abstraction de certains détails de cette syntaxe concrète, ceci pour faciliter la définition de la sémantique et pour accroître l'efficacité des calculs.
On supposera donc désormais que le programme introduit respecte bien les contraintes syntaxique de sorte que nous puissions le transformer en la structure plus abstraite décrite ci-dessous.
L'abstraction se porte essentiellement sur les quelques points cités ci-dessous :

- Les variables et les prédicats seront identifiés par un numéro et non plus par une chaîne de caractères.

  Ceci permet d'accéder à leur définition par un simple accès à un tableau, ce qui est à la fois très rapide et simple à implémenter (plus simple que d'implémenter une table *hash* par exemple).

  En plus, le fait que l'on numérote les variables, permet de distinguer des variables qui apparaissent dans des définitions de prédicat différentes, même si ce variables portent le même nom.
  Chaque variable étant identifiée de façon univoque par son numéro, on peut utiliser ce numéro pour accéder à la définition de son domaine et à la cardinalité de ce dernier.
  On remarque aussi que la façon dont on numérote les variables a des répercussions pour la représentation des relations. En effet, dans cette représentation, les variables seront d'une certaine manière triées selon leurs numéros.

- Les constantes seront aussi remplacées par des numéros. Une constante reçoit le numéro i, si elle est la $(i+1)$-ième constante dans son domaine.

  - Ainsi, si on considère le domaine

    domain toto {a,b,c},

    on remplace, dans tout le programme, chaque occurence de a (resp. b,c) par 0 (resp. 1, 2).

  - De même, si on considère le domaine de type intervalle

    domain toto a[5..7],

    on remplace chaque occurence de $a[5]$ (resp. $a[6], a[7]$) par 0 (resp. 1,2).

- Une fois les constantes numérotées, on n'a, pour ce qui est des calculs, plus besoin de retenir les définitions des domaines et le typage des variables.
  Tout ce dont on doit se rappeler est le nombre de constantes dans le domaine de chaque variable, ce qui peut être réalisé par une seule fonction *GetCardVar*.

- On va remplacer certaines structures syntaxiques par d'autres avec la même sémantique. Par exemple si on considère un domaine

  domain toto a[1..5],

  on va remplacer :

  - X in $a[3..5] \longmapsto X$ in $\{a[3], a[4], a[5]\}$
  - not X in $\{a[3], a[4], a[5]\} \longmapsto X$ in $\{a[1], a[2]\}$
  - $X = a[1] \longmapsto X$ in $\{a[1]\}$
  - not $X = Y \longmapsto X <> Y$

  Ceci facilite le traitement du programme β, tout comme le nombre de cas sémantiques à considérer qui s'en trouve réduit.

6.2 La syntaxe abstraite

Sur ce niveau plus abstrait, on considère qu'un programme est composé de

- Un ensemble de symboles $\{X_0, \ldots, X_\mu\}$[8], qui représentent les variables du programme.
- Une fonction GetCardVar : $\{X_0, \ldots, X_\mu\} \mapsto \mathbb{N}_0$, qui associe à chaque variable la cardinalité de son domaine.
- Un ensemble de symboles $\{p_0, \ldots, p_\nu\}$[8], qui représentent les prédicats du programme.
- Une séquence de $\nu + 1$ définitions de prédicat.

Avant de spécifier ce qu'est une définition de prédicat, nous devons considérer les sous-formules :

- Une **appartenance** a la forme
  $$X \in \{a_1, \ldots, a_n\},$$
  où X est une variable et les $a_i \in \mathbb{N}_0$.
- Une **comparaison de variables** a la forme
  $$X = Y \text{ ou } X \neq Y,$$
  où X et Y sont des variables.
- Une **contrainte** est soit une appartenance, soit une comparaison de variables.
- Une **conjonction de contraintes** a la forme
  $$c_1 \wedge \ldots \wedge c_n,$$
  où $c_1, \ldots, c_n$ sont des contraintes.
- Un **paramètre effectif** est soit une variable, soit une constante dans $\mathbb{N}_0$.
- Un **appel de prédicat** a la forme
  $$p(e_1, \ldots, e_n),$$
  où p est un prédicat et les e_i sont des paramètres effectifs.
- Un **litéral** a la forme
  $$A \text{ ou } \neg A,$$
  où A est un appel de prédicat.
- Une **conjonction de litéraux** a la forme
  $$l_1 \wedge \ldots \wedge l_n,$$
  où $l_1, \ldots, l_n$ sont des litéraux
- Une **conjonction** peut avoir une des trois formes suivantes :
  $$\begin{cases} \textit{contraintes} \wedge \textit{literaux} \\ \textit{contraintes} \\ \textit{literaux} \end{cases}$$
  où *contraintes* est une conjonction de contraintes et *literaux* une conjonction de literaux.
- Une **conjonction quantifiée** a la forme
  $$\exists X_{i_1}, \ldots, X_{i_n} : C,$$
  où C est une conjonction et les X_{i_j} sont des variables.
- Une **disjonction** a la forme
  $$C_1 \vee \ldots \vee C_n,$$
  ou $C_1, \ldots, C_n$ sont des conjonctions quantifiées.

Enfin, la **définition d'un prédicat** a la forme suivante :

$$p(X_1, \ldots, X_n) := D,$$

où p est un prédicat, les X_i sont des variables et D est une disjonction.

[8] On remarque que dans notre implémentation les symboles que l'on utilise pour identifier les variables et les prédicats seront des entiers. Sur ce niveau-ci on peut rester général et considérer des symboles quelconques, l'important c'est que le formalisme permet que l'on introduise des numéros

6.3 Le passage de la syntaxe concrète à la syntaxe abstraite

Le passage de la syntaxe concrète à la syntaxe abstraite devrait être assez évidente.

Une chose qui n'est peut-être pas évidente pour quelqu'un qui se connaît mal en programmation logique, est que dans une conjonction quantifiée, les variables que l'on quantifie sont toutes les variables qui apparaissent dans la conjonction et qui ne sont pas des paramètres du prédicat pour lequel la conjonction est définie.

Une autre remarque à faire est que l'on suppose que les variables sont identifiées par leur symbole. Concrètement, on suppose que :

- une variable n'apparaît pas dans deux définitions de prédicat distinctes
- si une variable se trouve quantifiée dans une conjonction quantifiée, elle n'apparaît nul part ailleurs dans le programme

7 La sémantique du β-langage

7.1 Introduction

Le but de ce chapitre est de définir la sémantique (ou *le sens*) d'un programme P écrit dans le β-langage. Considérons à cet égard la définition suivante :

Soit P un programme avec comme variables $\{X_0, \ldots, X_\mu\}$, fonction de cardinalité *GetCardVar* et prédicats $\{p_0, \ldots, p_\nu\}$.

Une **interprétation** de P est une fonction

$$B : \{p_0, \ldots, p_\nu\} \rightarrow \mathcal{SB}$$

qui associe à chaque prédicat p du programme une bi-relation

$$B_p =< \mathrm{Pos}_p, \mathrm{Neg}_p > .$$

On suppose que cette interprétation respecte les définitions des prédicats, c'est-à-dire que si la définition de p a la forme

$$p(X_{i_1}, \ldots, X_{i_n}) := D,$$

alors B_p a les étiquettes

$$\{X_{i_1}, \ldots, X_{i_n}\}$$

et le domaine

$$U_1 \times \ldots \times U_n \text{ où } U_j = \{0, \ldots, \mathrm{GetCardVar}(X_{i_j})\}.$$

Notons l'ensemble des interprétations de P par Ω_p.

Intuitivement la sémantique d'un programme P est une interprétation B de P telle que pour chaque prédicat p, Pos_p (resp. Neg_p) est l'ensemble de tous les tuples pour lesquels on peut déduire que p est vrai.

Cette interprétation est obtenue en partant de l'interprétation vide, qui associe chaque prédicat p à la birelation $< \emptyset, \emptyset >$ et modélise ainsi, qu'au début, on ne sait pas du tout où le prédicat est vrai ou faux. Ensuite, on va itérativement améliorer cette interprétation jusqu'au moment où l' on aura déduit tout ce qu'il y a à déduire.

Nous y reviendrons au paragraphe (7.4), mais d'abord on va s'attarder sur la sémantique des sous-formules, en commençant par les contraintes.

7.2 La sémantique d'une contrainte

La sémantique d'une contrainte ou d'une conjonction de contraintes est tout simplement la relation des tuples où la contrainte ou la conjonction de contraintes est satisfaite.
Nous pouvons décrire ceci par la fonction *ContrainteToRel* qui associe une relation à une contrainte et qui est définie comme suit :

- Pour une **appartenance** :

$$\mathrm{ContrainteToRel}(X \in \{a_1, \ldots, a_n\}) = R$$
$$\text{où } R = \{\{X : a\} \mid a \in \{a_1, \ldots, a_n\}\}$$

- Pour une **comparaison de variables** :

$$\mathrm{ContrainteToRel}(X = Y) = R$$
$$\mathrm{ContrainteToRel}(X \neq Y) = \mathrm{Not}(R)$$
$$\text{où } R = \{\{X : a, Y : a\} \mid a \in \{0, \ldots, \mathrm{GetCardVar}(X)\}\}$$

Nous en déduisons la fonction *ContraintesToRel* qui associe une **conjonction de contraintes** à sa relation correspondante :

Soit $C_1, \ldots, C_n$ des contraintes, ContrainteToRel$(C_i) = R_i$, alors :

ContraintesToRel$(C_1 \wedge \ldots \wedge C_n) = R_1 \wedge \ldots \wedge R_n$

7.3 La sémantique d'une sous-formule

La sémantique d'une sous-formule est un peu moins évidente.
Selon l'approche de [4], la sémantique d'une sous-formule f est décrite par un opérateur $\mathcal{F}$ qui prend une interprétation $B \in \Omega_P$ et qui renvoie une birelation $\mathcal{F}_B(f)$.

Intuitivement ceci représente que si on a une interprétation $B \in \Omega_P$, c'est-à-dire une connaissance partielle des prédicats de P, on peut en déduire des tuples pour lesquels la sous-formule sera vraie (resp. fausse), donc une birelation.

Formellement l'opérateur $\mathcal{F}_B$ est défini comme suit :

- Pour un **appel de prédicat** :

  Si la définition du prédicat appelé a la forme
  $$p(X_{i_1}, \ldots, X_{i_n}) := D$$
  et si $B_p =< \text{Pos}_p, \text{Neg}_p >$, alors :

  $\mathcal{F}_B(p(e_1, \ldots, e_n)) =< R, S >$
  où $R = \text{Substituer}(\text{Pos}_p, [X_{i_1} \rightarrow e_1, \ldots, X_{i_n} \rightarrow e_n])$
  $S = \text{Substituer}(\text{Neg}_p, [X_{i_1} \rightarrow e_1, \ldots, X_{i_n} \rightarrow e_n])$
- Pour un **littéral** :

  Soit A un appel de prédicat et $\mathcal{F}_B(A) =< \text{Pos}, \text{Neg} >$, alors :

  $\mathcal{F}_B(\neg A) =< \text{Neg}, \text{Pos} >$
- Pour une **conjonction de littéraux** :

  Soit $L_1, \ldots, L_n$ des littéraux, $\mathcal{F}_B(L_i) =< \text{Pos}_i, \text{Neg}_i >$, alors :

  $\mathcal{F}_B(L_1 \wedge \ldots \wedge L_n) =< \text{Pos}_1 \wedge \ldots \wedge \text{Pos}_n, \text{Neg}_1 \vee \ldots \vee \text{Neg}_n >$
- Pour une **conjonction** :

  Soit C une conjonction de contraintes, L une conjonction de littéraux :
  - $\mathcal{F}_B(L)$ est déjà défini.
  - $\mathcal{F}_B(C) =< R, \text{Not}(R) >$
    si $R = \text{ContraintesToRel}(C)$
  - $\mathcal{F}_B(C \wedge L) =< \text{Pos}_1 \wedge \text{Pos}_2, \text{Neg}_1 \vee \text{Neg}_2 >$
    si $\text{Pos}_1 = \text{ContraintesToRel}(C)$,
    $\text{Neg}_1 = \text{Not}(\text{ContraintesToRel}(C))$,
    $\mathcal{F}_B(L) =< \text{Pos}_2, \text{Neg}_2 >$
- Pour une **conjonction quantifiée** :

  Soit C une conjonction, $\mathcal{F}_B(C) =<$ Pos,Neg $>$:

  $\mathcal{F}_B(\exists X_{i_1}, \ldots, X_{i_n} : C) =< R, S >$,
  où $R = \text{IlExiste}(\text{Pos}, [X_{i_1}, \ldots, X_{i_n}])$
  $S = \text{PourTout}(\text{Neg}, [X_{i_1}, \ldots, X_{i_n}])$
- Pour une **disjonction** :

Soit $C_1, \ldots, C_n$ des conjonctions quantifiées, $\mathcal{F}_B(C_i) =< \mathrm{Pos}_i, \mathrm{Neg}_i >$, alors :

$\mathcal{F}_B(C_1 \vee \ldots \vee C_n) =< \mathrm{Pos}_1 \vee \ldots \vee \mathrm{Pos}_n, \mathrm{Neg}_1 \wedge \ldots \wedge \mathrm{Neg}_n >$

7.4 La sémantique d'un programme

Nous pouvons maintenant définir l'opérateur XB_P, qui prend une interprétation $B \in \Omega_P$ comme approximation de la sémantique du programme P et renvoie l'approximation suivante.
Cet opérateur

$$\mathrm{XB}_P : \Omega_P \to \Omega_P :< B_{p_0}, \ldots, B_{p_n} > \mapsto < B'_{p_0}, \ldots, B'_{p_n} >$$

est défini comme suit :

Suppose que la définition du prédicat p ait la forme

$$p(X_{i_1}, \ldots, X_{i_n}) := D,$$

alors $B'_p = \mathcal{F}_B(D)$.

On peut démontrer (cfr. [4]) que cet opérateur est toujours monotone. Dans le cas des domaines finis que nous considérons, il est même continu et on peut donc définir la sémantique du programme P comme le plus petit point fixe de XB_P.

7.5 La réponse à une requête

Nous avons défini la sémantique d'un programme P comme une interprétation $B \in \Omega_P$.
En général l'utilisateur ne sera pourtant pas intéressé à connaître les birelations pour chacun des prédicats qu'il a défini. On peut supposer qu'il va utiliser notre langage de la même façon que l'on utilise des autres langages logique comme par exemple PROLOG : c'est-à-dire qu'il va écrire son programme et puis faire une requête sur la partie de la solution qui l'intéresse.
Dans cette première version du programme nous admettons deux types de requêtes :

Soit P un programme, une **requête** sur P a la forme

$?p$ ou $?\neg p$

où p est un prédicat du programme.

Le résultat d'une requête peut alors être spécifié comme suit :

Soit P un programme, $B \in \Omega_P$ la sémantique de P, p un prédicat de P.

La **réponse à la requête** $?p$ (resp. $?\neg p$) est la relation Pos_p (resp. Neg_p) où $<\mathrm{Pos}_p,\mathrm{Neg}_p > = \mathrm{B}_p$.

7.6 Exemple

A titre d'exemple nous allons dans ce paragraphe calculer à la main la sémantique d'un petit programme dans le β-langage.
L'exemple que nous allons considérer est le suivant[9] :

$$\begin{cases} \mathrm{pere(X,Y)} & := \quad X = a \;\&\; Y = b \mid X = b \;\&\; Y = c\,; \\ \mathrm{ancetre(X,Y)} & := \quad \mathrm{pere}(X,Y) \mid \mathrm{ancetre}(X,Z) \;\&\; \mathrm{pere}(Z,Y)\,; \end{cases}$$

Une représentation abstraite de ce programme est par exemple :

$$\begin{cases} \mathrm{pere}(X_0, X_1) := X_0 \in \{0\} \wedge X_1 \in \{1\} \vee X_0 \in \{1\} \wedge X_1 \in \{2\}\,; \\ \mathrm{ancetre}(X_2, X_3) := \mathrm{pere}(X_2, X_3) \vee \exists X_4 : \mathrm{ancetre}(X_2, X_4) \wedge \mathrm{pere}(X_4, X_3)\,; \end{cases}$$

Une interprétation de ce programme a la forme

$$(< \mathrm{Pos}_{\mathrm{pere}}, \mathrm{Neg}_{\mathrm{pere}} >, < \mathrm{Pos}_{\mathrm{ancetre}}, \mathrm{Neg}_{\mathrm{ancetre}} >).$$

[9] Cet exemple a déjà été examiné de nombreuses fois : voir entre autres la page 9. Toutefois, la version utilisée ici en encore simplifiée.

Elle représente pour nos deux prédicats ancetre et p_2 l'ensemble de tuples où on sait déjà que ces prédicats sont vrais (resp. faux).

L'opérateur XB_P qui applique une interprétation du programme à une nouvelle, prend maintenant la forme :

$$XB_P : (< Pos_{pere}, Neg_{pere} >, < Pos_{ancetre}, Neg_{ancetre} >) \mapsto (< Pos'_{pere}, Neg'_{pere} >, < Pos'_{ancetre}, Neg'_{ancetre} >),$$

où si on applique la définition de cet opérateur :

$$\begin{cases} Pos'_{pere} = \begin{array}{c|c} X_0 & X_1 \\ \hline 0 & 1 \\ 1 & 2 \end{array} \\ Neg'_{pere} = \begin{array}{c|c} X_0 & X_1 \\ \hline 0 & 0 \\ 0 & 2 \\ 1 & 0 \\ 1 & 1 \\ 2 & 0 \\ 2 & 1 \\ 2 & 2 \end{array} \\ Pos'_{ancetre} = IlExiste(\, Substituer(Pos_{pere}, [X_0 \to X_2, X_1 \to X_3]), [\,]) \vee \\ \qquad IlExiste(\, Substituer(Pos_{ancetre}, [X_2 \to X_2, X_3 \to X_4]) \wedge \\ \qquad\quad Substituer(Pos_{pere}, [X_0 \to X_4, X_1 \to X_3]), \\ \qquad\quad [X_4]) \\ Neg'_{ancetre} = PourTout(\, Substituer(Neg_{pere}, [X_0 \to X_2, X_1 \to X_3]), [\,]) \wedge \\ \qquad PourTout(\, Substituer(Neg_{ancetre}, [X_2 \to X_2, X_3 \to X_4]) \vee \\ \qquad\quad Substituer(Neg_{pere}, [X_0 \to X_4, X_1 \to X_3]), \\ \qquad\quad [X_4]) \end{cases}$$

La sémantique de P est le point fixe de cet opérateur.
Notre interpréteur la calculera à l'aide de l'algorithme évoqué au chapitre 4. Dans ce petit exemple, on peut la trouver à l'œil :

- Comme première approximation, on prend

$$\begin{cases} Pos^0_{pere} = FALSE \\ Neg^0_{pere} = FALSE \\ Pos^0_{ancetre} = FALSE \\ Neg^0_{pere} = FALSE \end{cases}$$

- Si on introduit ceci dans la partie droite de nos équations, on trouve comme nouvelle approximation :

$$\begin{cases} Pos^1_{pere} = \begin{array}{c|c} X_0 & X_1 \\ \hline 0 & 1 \\ 1 & 2 \end{array} \\ Neg^1_{pere} = \begin{array}{c|c} X_0 & X_1 \\ \hline 0 & 0 \\ 0 & 2 \\ 1 & 0 \\ 1 & 1 \\ 2 & 0 \\ 2 & 1 \\ 2 & 2 \end{array} \\ Pos^1_{ancetre} = FALSE \\ Neg^1_{ancetre} = FALSE \end{cases}$$

A partir de maintenant, les approximations de Pos_{pere} et Neg_{pere} ne changent plus, il ne reste plus qu'à itérer le calcul de Pos_{ancetre} et Neg_{ancetre}.

$\text{Pos}^2_{\text{ancetre}}$ =

X_2	X_3
0	1
1	2

$\text{Neg}^2_{\text{ancetre}}$ =

X_2	X_3
0	0
1	0
2	0

$\text{Pos}^3_{\text{ancetre}}$ =

X_2	X_3
0	1
0	2
1	2

$\text{Neg}^3_{\text{ancetre}}$ =

X_2	X_3
0	0
1	0
1	1
2	0
2	1

$\text{Pos}^4_{\text{ancetre}}$ =

X_2	X_3
0	1
0	2
1	2

$\text{Neg}^4_{\text{ancetre}}$ =

X_2	X_3
0	0
1	0
1	1
2	0
2	1
2	2

- La prochaine itération ne fait que confirmer les valeurs que l'on vient de trouver, on a donc atteint le point fixe.

Ayant calculé la sémantique, on voit que la réponse à la requête

? ¬ *pere*

est :

X_0	X_1
a	a
a	c
b	a
b	b
c	a
c	b
c	c

Et qu'à la requête

? *ancetre*

notre interpréteur répondra correctement :

X_2	X_3
a	b
a	c
b	c

7.7 Conclusions

On remarque que dans la β-sémantique, les informations négatives (comme Neg_{pere} et Neg_{ancetre}) sont inférées sur pied d'égalité avec les informations positives.

Ceci distingue la β-sémantique fondamentalement de, par exemple, la syntaxe de PROLOG où les informations négatives sont inférées par l'impossibilité d'inférer l'information positive (la négation par échec).

Cette approche originale permet qu'un interpréteur pour le β-langage réponde correctement à la requête

? $\neg$ pere

Là où, si l'on considère un programme PROLOG correspondant comme

```
{ pere(a,b).
  pere(b,c).
```

un interpréteur PROLOG répondra à la requête

? not p(X,Y)

par *non* ou *false*.

Ce résultat inattendu s'explique par le fait que pour répondre à cette requête, PROLOG va d'abord essayer de résoudre la requête

? p(X,Y).

Comme cette requête n'échoue pas (on trouve les solutions {X/a,Y/b} et {X/b,Y/c}), il ne peut pas inférer de solutions pour la requête

? not p(X,Y)

Il y a donc en PROLOG une très grande asymétrie entre les inférences positives et négatives.
La où la requête

? p(X,Y)

correspond en fait à la question

Est-ce que **il existe** des X, Y t.q. p(X,Y) ?

la requête négative

? not p(X,Y)

correspond à

Est-ce que **pour tout** X et Y : not p(X,Y) ?

C'est à cause de ce genre de contradictions que l'utilisation de la négation en PROLOG n'est pas chose évidente. Des précautions doivent être prises pour s'assurer, pour un programme concret, que la sémantique de la négation en PROLOG correspond bien à sa signification déclarative.
Concrètement on évitera en PROLOG d'utiliser des atomes négatifs qui ne seront, au moment de leur appel, pas complètement instanciés.

Même si on se limite à des constructions positives, notre β-langage est beaucoup plus déclaratif que PROLOG.
On voit facilement que la sémantique d'un programme en β-langage ne change pas si on change l'ordre de ses constituants.
Ainsi le petit programme que nous avons étudié est complètement équivalent au programme suivant :

$$\begin{cases} \text{ancetre}(X,Y) & := & \text{ancetre}(X,Z) \mid \text{pere}(X,Y) \,\&\, \text{pere}(Z,Y)\,; \\ \text{pere}(X,Y) & := & X = a \;\&\; Y = b \mid X = b \;\&\; Y = c\,; \end{cases}$$

En PROLOG ceci est loin d'être le cas.
Si on considère par exemple la requête

? ancetre(X,Y)

pour le programme

```
pere(a,b).
pere(b,c).
ancetre(X,Y) :- ancetre(X,Z), pere(Z,Y).
ancetre(X,Y) :- pere(X,Y).
```

l'interpréteur PROLOG bouclera sans nous fournir aucune solution.
Si on échange deux clauses et que l'on considère

```
pere(a,b).
pere(b,c).
ancetre(X,Y) :- pere(X,Y).
ancetre(X,Y) :- ancetre(X,Z), pere(Z,Y).
```

l'interpréteur PROLOG nous fournira bel et bien toutes les solutions, mais bouclera après.
La bonne solution est le programme

```
pere(a,b).
pere(b,c).
ancetre(X,Y) :- pere(X,Y).
ancetre(X,Y) :- pere(Z,Y), ancetre(X,Z).
```

pour lequel l'interpréteur s'arrête après avoir donné les solutions.
A cause du fait que PROLOG n'est pas complètement déclaratif, la rédaction d'un programme PROLOG n'est pas une chose facile.
La transformation d'une description logique en un programme PROLOG qui donne toutes les solutions et qui se termine, demande encore toute une analyse.
Notre β-langage par contre peut faire valoir ses titres de langage strictement déclaratif.

Après avoir chanté les louanges du β-langage, il faut quand même que l'on relativise un peu.

Il est vrai que PROLOG n'est pas complètement déclaratif et que l'implémentation de la négation n'est saine que sous certaines conditions. Mais imparfait comme il l'est du point de vue logique, comme langage de programmation il est presque tout aussi efficace que les langages impératifs.
Notre langage fait mieux au niveau logique, mais le fait à un coût très élevé.
On peut démontrer que si on considère des domaines infinis, l'opérateur XB_P qu'on a utilisé pour définir la sémantique n'est en général plus continu, ce qui fait que le point fixe ne pourra plus être approximé en un temps fini.
Déjà avec des domaines finis, on se heurte vite aux limites de notre interpréteur. Sur une machine avec 32M de RAM et une mémoire virtuelle de 16M, le problème des 11 reines n'est plus réalisable.
La question de savoir lequel des deux langages est le meilleur n'est donc pas nettement tranchable, mais dépendra toujours du genre de problème que l'on veut résoudre.

L'idéal serait en effet de développer un langage qui combine les points forts des deux langages.
C'est en effet avec cette aspiration que la β-sémantique a été développée.
Nous nous sommes d'emblée limité à son opérationnalisation pour le cas des domaines finis, mais en réalité la β-sémantique est beaucoup plus générique que ça.
Dans [4], on explique que dès qu'on limite l'ensemble des problèmes que l'on veut résoudre, d'une telle manière que pour cet ensemble restreint l'opérateur XB_P est continu, on peut procéder à l'opérationnalisation de la sémantique.
Si on prend un autre point de départ que le nôtre (par exemple si on restreint les situations dans lesquelles on peut utiliser la négation), on peut arriver à d'autres opérationnalisations qui sont plus proches de PROLOG ou même des langages fonctionnels.
On espère que, comme tous les langages ainsi construits partagent sur un niveau plus profond la même sémantique, on pourra ensuite procéder à leur intégration au sein d'un langage générique.

8 Un paradoxe du β-langage

En faisant des tests avec notre logiciel, nous nous sommes rendu compte que la programmation en β-langage ne donne pas toujours les résultats attendus.

Dans le temps qui nous reste, nous ne pouvons plus entamer une étude profonde du phénomène découvert, mais nous devons nous limiter à l'énoncé du problème ainsi qu'à une preuve mathématique du fait que, aussi inattendu que les résultats peuvent être, ils sont bien prescrits par la β-sémantique.

8.1 Enoncé du paradoxe

Considérons l'exemple ci-dessus :

```
domain {g,v,a};

nat(M,M') := nat_1(M,M') | M'=g;

nat_1(M,M') := nat(v,M'') & aux(M,M'',M');

aux(X,Y,Z) :=
        X = g & Z = g
     |  X = v & Y = g & Z = g
     |  X = v & Y = v & Z = a
     |  X = v & Y = a & Z = a
     |  X = a & Y = g & Z = g
     |  X = a & Y = v & Z = a
     |  X = a & Y = a & Z = a;

sub(M,M') :=
        M = g & M' = a
     |  M = v & M' = a;

nat_sub(M,M') := nat(M,M'') & sub(M',M'');

nat_libre(M,M') := nat(M,M') & not nat_sub(M,M');
```

Nous avons rencontré ce programme pour l'interprétation abstraite du programme PROLOG suivant :

```
nat(X1,X2) :- nat(X2), X1 = s(X1).
nat(X1,X2) :- 0.
```

dans le domaine abstrait **{ground,var,any}**.

Le prédicat *nat* exprime la relation qui existe entre le mode d'entrée M et le mode de sortie M' pour le prédicat PROLOG *nat*.

Dans le chapitre 24 nous reviendrons en détail sur la signification précise de ce programme. Pour l'instant seuls importent les résultats aux requêtes.

Ainsi, si on fait la requête
? nat,
notre interpréteur répondra par la table suivante :

M	M'
g	g
v	g
a	g

Dans ce cas particulier les trois lignes obtenues sont indépendantes, mais dans d'autres exemples dans le contexte de l'interprétation abstraite, nous étions souvent confronté avec des tables comme suit :

(M1 :g, M2 :g, M3 :g, M1' :g, M2' :g, M3' :g)

(M1 :g, M2 :g, M3 :v, M1' :g, M2' :g, M3' :g)
(M1 :g, M2 :g, M3 :v, M1' :g, M2' :g, M3' :a)

(M1 :v, M2 :a, M3 :v, M1' :g, M2' :g, M3' :g)
(M1 :v, M2 :a, M3 :v, M1' :g, M2' :v, M3' :v)
(M1 :v, M2 :a, M3 :v, M1' :a, M2' :a, M3' :a)

(M1 :v, M2 :v, M3 :a, M1' :g, M2' :g, M3' :g)
(M1 :v, M2 :v, M3 :a, M1' :g, M2' :a, M3' :a)
(M1 :v, M2 :v, M3 :a, M1' :a, M2' :a, M3' :a)

Comme nous l'expliquerons plus en détail au chapitre 24, il existe une relation de subsomption entre les lignes d'une telle table.

On dira qu'une ligne est subsummée par une autre si la seule différence qu'il existe entre ces deux lignes est qu'un nombre de constantes g ou v dans une des colonnes accentuées dans la première ligne deviennent a dans la deuxième ligne.

Ainsi dans l'exemple

- la deuxième ligne est subsummée par la troisième ligne
- la quatrième et la cinquième ligne sont subsummées par la sixième ligne
- la septième et huitième ligne sont subsummées par la neuvième ligne.

Pour notre application à l'interprétation abstraite, on s'intéresse seulement aux lignes qui ne sont subsummées par aucune autre ligne de la table.

Comme on l'a dit, tout ceci sera plus amplement expliqué au chapitre 24, mais cette petite introduction devrait suffir pour comprendre l'origine des prédicats *sub*, *nat_sub* et *nat_libre* dans notre petit exemple.

- Le prédicat *sub* exprime que les constantes g et v sont subsummées par la constante a.
- On appelle un couple (M, M') subsummé par la table *nat*, s'il existe un couple (M, M'') dans *nat* tel que M' est subsummé par M.
  Ceci est exprimé par le prédicat *nat_sub*.
- Une ligne (M, M') de la table *nat* est appelée "libre" si elle n'est pas subsummée par la table.
  Ceci est exprimé par le prédicat *nat_libre*.

Etant donné que la table *nat* ne contient pas la constante a dans la colonne accentuée, on s'attendait à ce que le résultat à la requête

? nat_libre

soit simplement la même table...

Imaginez notre étonnement quand le résultat est apparu :

M	M'
g	g

Un étonnement d'autant plus grand que, après l'ajout des lignes suivantes au programme

```
nat_test(M,M') :=
      M=g & M'=g
    | M=v & M'=g
    | M=a & M'=g;
```

```
nat_test_sub(M,M') := nat_test(M,M'') & sub(M',M'');

nat_test_libre(M,M') := nat_test(M,M') & not nat_test_sub(M,M');
```

la réponse à la requête

? nat_test_libre

fût bel et bien le

M	M'
g	g
v	g
a	g

attendu.

Malheureusement il ne s'agît ici pas d'une erreur dans notre programme β, ni d'un bug dans notre interpréteur... Comme nous le démontrerons dans le paragraphe suivant, les résultats obtenus sont effectivement les solutions décrites par la β–sémantique...

L'origine du paradoxe apparant est que la β-sémantique ne calcule pas un modèle du programme, mais seulement l'ensemble d'informations (positives et négatives) qui sont valides dans **tous** les modèles du programme.

Bien que parfaitement compréhensible d'un point de vue mathématique il est clair que des situations comme décrites ici sont très gênantes pour le statut du β-langage comme langage de programmation.

Sans doute pourra-t-on formuler des conditions sous lesquelles ce genre de paradoxes peuvent être évités, mais cela ne semble pas un problème évident.
Nous nous limiterons donc à une explication théorique du paradoxe rencontré.

8.2 Explication du paradoxe

Rappelons que la réponse à la requête

? nat

était :

$\text{Pos}_{\text{nat}} =$

M	M'
g	g
v	g
a	g

De la même façon on trouve par la requête

? not nat

que :

$\text{Neg}_{\text{nat}} =$

M	M'
g	v
g	a
v	v
a	v

On observe que, apparamment

$$\text{Pos}_{\text{nat}} \cup \text{Neg}_{\text{nat}} \neq \text{TRUE}$$

et que plus précisément les tuples

M	M'
v	a
a	a

n'appartiennent à aucun des deux ensembles.

La raison de ceci est facile à comprendre une fois qu'on se rend compte que le système d'équations logiques :

$$\begin{cases} \text{nat}(M, M') & \Leftrightarrow & \text{nat1}(M, M') \vee M' = g \\ \text{nat}(M, M') & \Leftrightarrow & \exists M'' : \text{nat}(v, M'') \wedge \text{aux}(M, M'', M') \\ \text{aux}(X, Y, Z) & \Leftrightarrow & \ldots \end{cases}$$

admet (exactement) deux interprétations dans le domaine $\{g, v, a\}$.

- Une première où

nat = nat1 =

M	M'
g	g
v	g
a	g

et où *nat(v,a)* et *nat(a,a)* sont faux.

- Et une deuxième où

nat = nat1 =

M	M'
g	g
v	g
a	g
v	a
a	a

et où *nat(v,a)* et *nat(a,a)* sont vrais.

Dans notre article de référence [4] il est démontré que Pos_{nat} (resp. Neg_{nat}) est l'ensemble de tuples pour lesquels la relation *nat* est vraie (resp. fausse) dans chaque modèle de la théorie. Ceci correspond effectivement aux valeurs que nous avons obtenues.

Si on étend maintenant la théorie avec les formules

$$\begin{cases} \text{sub}(M, M') & \Leftrightarrow & M = g \wedge M' = a \vee M = v \wedge M' = a \\ \text{nat\_sub}(M, M') & \Leftrightarrow & \exists M'' : \text{nat}(M, M'') \wedge \text{sub}(M', M'') \\ \text{nat\_libre}(M, M') & \Leftrightarrow & \text{nat}(M, M') \wedge \neg\, \text{nat\_sub}(M, M') \end{cases}$$

On trouve deux modèles :

- Un premier où

nat = nat1 =

M	M'
g	g
v	g
a	g

nat_sub = $\emptyset$

nat_libre =

M	M'
g	g
v	g
a	g

– Et un deuxième où

nat = natl =

M	M'
g	g
v	g
a	g
v	a
a	a

nat_sub =

M	M'
v	g
v	v
a	g
a	v

nat_libre =

M	M'
g	g
v	a
a	a

Si l'on sait que Pos_{nat_libre} est l'ensemble de tuples où la relation *nat_libre* est vraie dans *chaque* modèle de la théorie, on retrouve effectivement le résultat inattendu :

Pos_{nat_libre} =

M	M'
g	g

On comprend maintenant aussi la raison profonde du paradoxe :

Tout le problème provient du fait que la β-sémantique ne calcule pas de modèle de la théorie proposée (ce qui dans notre exemple ne serait d'ailleurs pas possible puisque la théorie en question n'admet pas de modèle minimal), mais calcule au lieu de cela l'ensemble des informations (possitives et négatives) qui sont valides dans **tous** les modèles de ladite théorie.

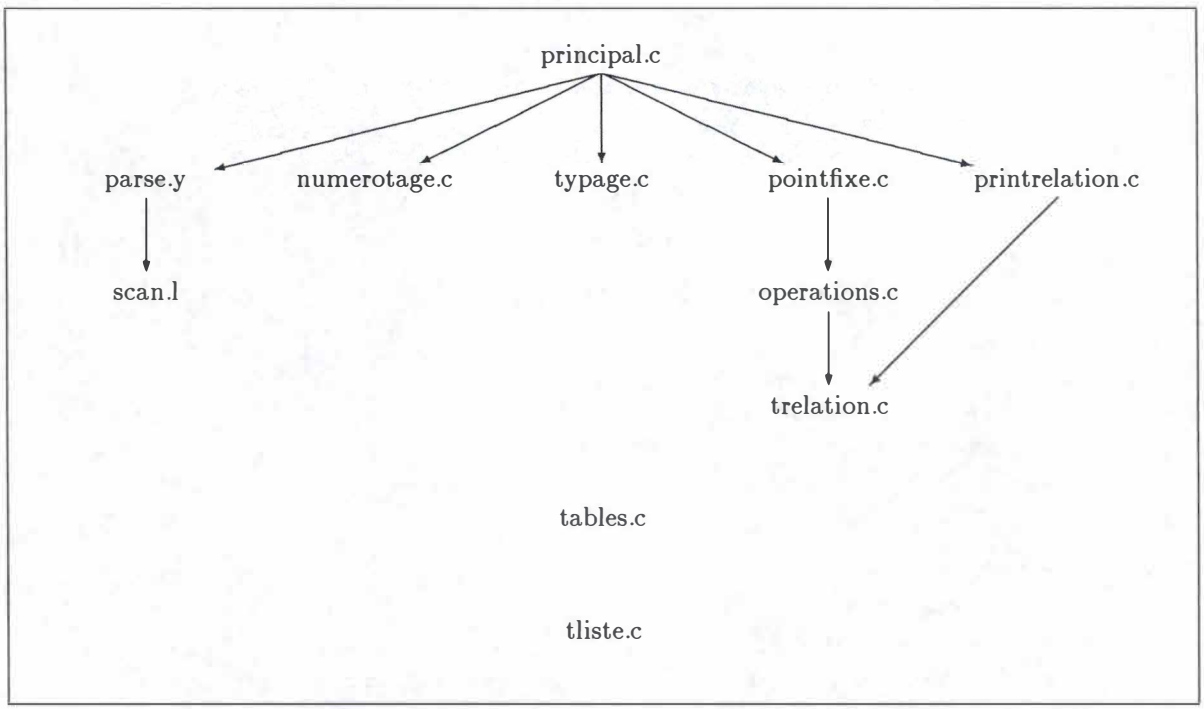
Comme on vient de le voir, ceci peut parfois résulter en une perte d'information tout à fait surprenante.

Une étude plus approfondie de ce phénomène semble s'imposer : Comment l'utilisateur peut-il être sûr qu'une programmation modulaire telle que nous l'avons essayée (d'abord calculer la table *nat* et ensuite écrire un nouveau prédicat pour enlever les lignes subsummées dans cette table) ne résultera pas en une perte de certaines solutions ?

Deuxième partie
Architecture

9 Présentation globale

Notre logiciel est structuré en onze modules comme indiqué sur le graphe ci-dessous.



Les flèches correspondent à une relation d'utilisation.
En plus, il existe une relation chronologique entre les modules sur la deuxième ligne.

Les deux modules en bas de figure sont vraiment des modules de base, ils sont utilisés un peu partout dans le programme et c'est la raison pour laquelle on n'a pas indiqué de relation d'utilisation pour ces deux-là.

Notre logiciel fonctionne essentielement en trois phases :

- Dans une première phase dite de parsage, notre interpréteur va vérifier si le programme à interpréter respecte bien toutes les contraintes syntaxiques évoquées au chapitre (5).
  Si ceci n'est pas le cas, il arrête en affichant un message d'erreur qui donne une description précise de l'erreur rencontrée (cfr. paragraphe (5.4) et (5.5)).
  Si le programme β respecte bien toutes les contraintes, sa forme textuelle est codée dans des types syntaxiques qui mettent en avant la structure syntaxique du programme.
  Ces types seront stockés dans des tables syntaxiques gérées par le module *tables.c* et accessibles partout dans notre logiciel.
  Le parsage est réalisé en quatre étapes, ce qui se réflete dans la présence de quatre modules :
  - Le module **scan.l** permet d'extraire les *tokens* du β-programme, c'est à dire les "mots" du β-langage (comme par exemple les noms de variable ou de prédicat, les mots-clefs comme *domain* ou *in*, les symboles comme " :=" ou "&", ...).
  - Le module **parse.y** peut ensuite vérifier si le programme source respecte bien les règles de production de la syntaxe concrète, elle peut aussi vérifier certaines contraintes de plus haut niveau comme le fait que les noms des prédicats sont uniques, ou que chaque domaine ou constante doit être déclaré avant d'être utilisé. En faisant ceci elle peut renvoyer les messages d'erreurs 3 à 14 (cfr. paragraphe (5.4)).
  - Le module **numerotage.c** peut ensuite vérifier si tous les prédicats appelés sont déclarés quelque part dans le programme source et que son arité est bien respecté.
    Si ceci n'est pas le cas, le message d'erreur 15 ou 16 est renvoyé.

On remarque que cette vérification n'est pas faite par *parse.y* puisque la déclaration d'un prédicat peut se trouver après son utilisation.

En plus de cette vérification, ce module a aussi comme fonction de remplacer les noms des prédicats et des variables par des numéros. Comme on l'a expliqué au paragraphe (6.1) ceci permettra un traitement plus efficace par la suite.

- Le module **typage.c** essaie de déterminer le type de chaque variable et de vérifier si tout le programme est libre de conflits de type.

  Si le type d'une variable ne peut être déterminé ou si un conflit de type est détecté, un des messages d'erreur 16 à 19 est renvoyé.

- La deuxième phase est la phase de calcul, où on va calculer la réponse à une requête comme définie au paragraphe (7.5).

  Cette fonctionalité et offerte par le module **pointfixe.c**.

- La troisième et dernière phase consiste alors en l'affichage du résultat du calcul sous un format compréhensible par l'utilisateur.

  Il s'agît d'afficher la relation obtenue (et qui est représentée de façon interne comme un arbre) sous forme de table et de remplacer les numéros des constantes et des variables par leurs noms.

  Cette fonctionalité est offerte par le module **printrelation.c**.

Les cinq modules restants sont les suivants :

- Le module **principal.c** est le module coordinateur de notre interpréteur, il contient une seule fonction *main* qui après avoir fait quelques initialisations, appelle les différentes fonctions pour le parsage, le calcul et l'affichage comme décrit ci-dessus.
- Les modules **trelation.c** et **operations.c**, implémentent ensemble les relations et les opérations relationnelles comme définies au chapitre (2).

  Les opérations relationnelles proprement dites sont implémentées dans le module *operations.c*.

  Le module *trelation.c* est de plus bas niveau et se limite à implémenter le type abstrait que l'on utilisera pour représenter les relations en mémoire.

  Nous avons choisi de les représenter comme des arbres et de nous organiser pour que, si deux relations ont un sous-arbre en commun, ce sous-arbre ne sera stocké en mémoire qu'une seule fois (voir partage des nœuds, page 69).
- Le module **tables.c** ne fait que gérer les tables dans lesquelles les informations syntaxiques sur le programme à interpréter seront stockées.
- Le module **tliste.c** finalement, implémente un type pour les listes génériques, avec des spécialisations pour les listes d'entiers et les listes de chaînes de caractères.

Dans le reste de ce chapitre nous reviendrons plus en détail sur chacun de ces modules.

10 La représentation des relations

module trelation.c
interface : **GetLabel** : TRelation → Integer **GetFils** : TRelation → TRelation [ ] **TRUE**, **FALSE** ∈ TRelation **InitRelationTable** : Integer → **CreateRelation** : Integer × TRelation [ ] → TRelation **CopyRelation** : TRelation → TRelation **DestroyRelation** : TRelation → **PrintRelationTable** : Boolean →
utilise : GetCardVar de tables.c

10.1 Ordre des variables

Comme il a déjà été expliqué auparavant, nous manipulons des relations étiquetées. Par convention, nous étiquetons par des noms de variables provenant des déclarations que l'utilisateur a écrites dans son code. Toutefois, pour ne pas être esclave de la longueur d'un nom de variable ou complexifier le code de l'interpréteur en comparant sans cesse des noms de variables, nous avons opté pour un "rebaptême" de l'ensemble des variables du code de l'utilisateur par des nombres entiers.

De plus, dans les opérations sur les relations, il est infiniment pratique de définir un ordre sur les nœuds des arbres représentant les relations. Renuméroter les variables par des entiers amène cela de façon très naturelle. Il est également très facile de retrouver le numéro du domaine auquel une variable appartient ainsi que le nombre de valeurs que peut prendre une variable[10](cardinalité du domaine).

Dans un but de lecture plus aisée, nous allons considérer un exemple. Considérons le programme suivant :

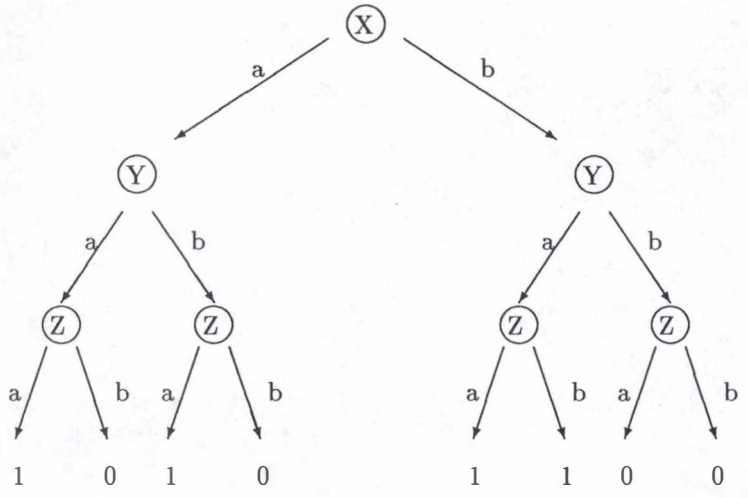
$$\begin{cases} \texttt{domain } \{a,b\}; \\ p(X,Y,Z) \;:=\; X=a \;\wedge\; Z=a \\ \qquad\qquad\;\; |\;\; X=b \;\wedge\; Y=a; \end{cases} \tag{5}$$

L'ensemble des tuples vérifiant l'énoncé du prédicat est le suivant :

$$\{(x,y,z) \dagger\, x,y,z \in \text{ domaine par défaut}\} = \{(a,a,a),(a,b,a),(b,a,a),(b,a,b)\}$$

[10] Le lecteur peut aller voir la section 6.2 page 46 et le chapitre 7 page 49 où l'on parle de la fonction **GetCardVar** permettant cette facilité.

Dans l'implémentation, cet ensemble est modélisé par un arbre :



où une feuille terminale 1 indique un tuple appartenant à la relation, et 0 un tuple n'appartenant pas à la relation.
Les règles de construction d'un arbre tiennent sur les doigts de la main. Pour fixer les idées, prenons $(X_i)_{i=1}^{n}$, l'ensemble des variables de l'en-tête d'un prédicat $p(X_1, \ldots, X_n)$ où l'indice correspond à l'ordre de déclaration de la variable dans ledit en-tête.

1. Le nœud racine de l'arbre est associé à X_1. Il possède $card(X_1)$ fils[11] ;
2. Un nœud terminal est un élément *vrai* ou *faux* ;
3. La variable associée à un nœud père d'un nœud terminal est X_n. Il possède $card(X_n)$ fils ;
4. Un nœud situé à une profondeur i est associé à la variable X_i et possède $card(X_i)$ fils (la racine de l'arbre est de niveau 1) ;
5. Chaque fils renseigne un nouveau nœud de profondeur supérieur, sauf en ce qui concerne les nœuds du dernier niveau de l'arbre qui renseignent une valeur *vraie* ou *fausse*.

Implicitement, la déclaration des variables dans l'en-tête allant de gauche à droite, on les renumérote en pratique en respectant ce sens. Ainsi, les variables vont en croissant au fur et à mesure que l'on pénètre profondément dans l'arbre. Dans ces règles, $card(X)$ représente la cardinalité d'une variable, c'est-à-dire le nombre d'éléments du domaine auquel elle appartient, ou encore le nombre de valeurs différentes qu'elle peut prendre. La définition du prédicat influe donc sur la représentation de la relation :

- d'abord parce que la définition de la relation dépend du prédicat ;
- ensuite parce que l'ordre des variables dans l'en-tête du prédicat modèle l'arbre représentant la relation.

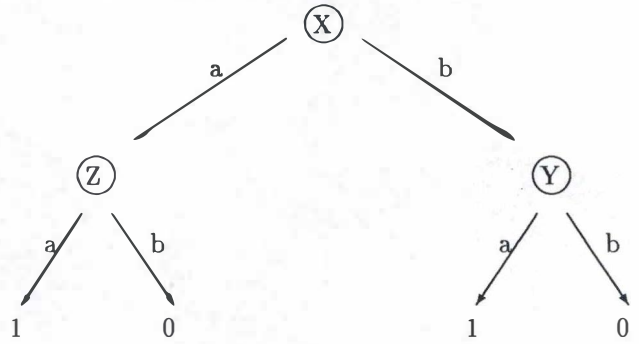
Dans le code de l'interpréteur, chaque prédicat reçoit deux relations de ce genre afin de modéliser la birelation qui caractérise ce prédicat. La sémantique est donc respectée sans faux-fuyant. Plus précisément, chaque prédicat reçoit au départ une birelation vide $< Pos, Neg >=< \emptyset, \emptyset >$ qui grossira au fur et à mesure de l'avancement de l'algorithme, cernant de plus en plus de valeurs du domaine.
On remarquera qu'une feuille terminale 1 dans Pos signifie qu'un tuple est vrai pour le prédicat, tandis qu'une feuille terminale 1 dans Neg signifiera qu'un tuple est faux pour le prédicat.

10.2 Représentation avec omission de variables

L'exemple précédemment introduit présente quelques propriétés. Par exemple, dans la partie gauche de l'arbre, les deux braches "Z" sont identiques. Ou encore, dans la partie droite, les branches "Z" ont les mêmes fils. Ces remarques permettent d'introduire la *représentation avec omission de variables*.

[11] C'est encore la fonction **GetCardVar** qui permet cette implémentation. Voir section 6.2 et chapitre 7

Plutôt que de toujours représenter un arbre dans sa totalité, on décide de synthétiser les informations redondantes. L'arbre de l'exemple devient alors :



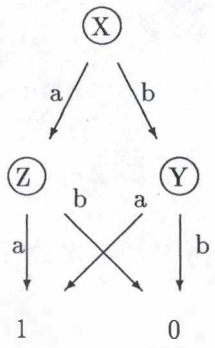
Ainsi, à gauche, le nœud "Y" a disparu. A droite, il n'y a plus de nœud "Z". Cette transformation obéit à la règle suivante :

Lorsque tous les fils sont identiques, ils sont synthétisés en un seul objet et le nœud est omis.

Cette façon de procéder a l'avantage de condenser l'information. De plus, elle ne pose pas de problème de lecture de l'arbre pourvu que les variables soient disposées selon l'ordre imposé par le prédicat (les nœuds de l'arbre restent ordonnés selon l'étiquetage qui a précédé). Ceci signifie qu'une variable omise dans une branche peut dès lors prendre n'importe quelle valeur du domaine auquel elle appartient. Enfin, diverses manipulations de relations restent permises sans toutefois compliquer trop les algorithmes.

10.3 Partage des nœuds

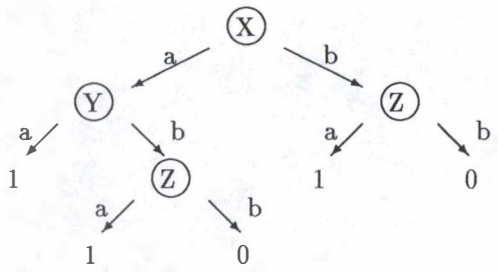
Afin d'économiser encore plus de place, certaines parties de l'arbre qui se ressemblent ne seront mémorisées qu'une seule fois. Dans l'exemple qui a été donné, on peut imaginer un représentation encore plus succinte :



Cette transformation supplémentaire permet de rester très synthétique. Elle peut même se réaliser à des niveaux différents de l'arbre. Ainsi, considérons la relation suivante qui a déjà subi une omission de

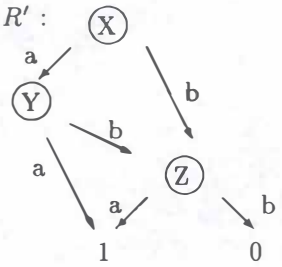
variable ("Y" à droite dans l'arbre) :

R :



On y remarque immédiatement que les nœuds étiquetés par "Z" renseignent des sous-arbres identiques. Condenser cet arbre avec partage de nœuds amène à la représentation suivante :

R devient R' :



qui de toute façon modélise le même ensemble de tuples, à savoir

$\{(a,a,a),(a,a,b),(a,b,a),(b,a,a),(b,b,a)\}$

10.4 Technique du hashing

10.4.1 Principe du hashing

Le stockage d'information peut utiliser deux philosophies différentes. La première consiste à ajouter les informations nouvelles au bout de celles déjà stockées ; c'est le cas du fichier séquentiel pur. La seconde consiste à réserver d'avance une certaine quantité de place, puis d'y stocker les éléments au fur et à mesure de leur arrivée ; il s'agit du hashing. Cette seconde technique peut accélérer énormément l'accès aux données si quelques dispositions élémentaires sont prises.
Ainsi, l'endroit où est stocké un élément est fonction de ce qui caractérise ou identifie cet élément. Ce fait permet de retrouver très facilement les informations qui sont jointes à cet élément dès lors que l'on dispose de la clé qui l'identifie car un simple calcul suffit.
Par exemple, considérons un petit livre d'adresses de cent personnes. Supposons que l'ensemble constitué du nom et du prénom permet toujours de retrouver une et une seule personne. Le stockage peut procéder comme suit :

- chaque lettre, considérée sans accent, est assimilée à son rang dans l'alphabet ;
- les traits d'union, blancs et apostrophes n'ont aucune valeur ;
- le nombre associé à une personne vient de l'addition de chaque nombre associé aux lettres des nom et prénom ;
- si le résultat dépasse cent, on ne retient que les deux derniers chiffres.

En appliquant ce principe à *Jean Dupont*, on trouve $10+5+1+14+4+21+16+15+14+20$, soit 120 qui est assimilé à 20. Pour obtenir ses coordonnées complètes, il suffira d'aller voir la page vingt du petit livre d'adresses. Si lors du stockage, cette page était déjà occupée, on aura pris soin d'écrire les informations à la première page libre qui suit afin d'éviter le recouvrement. C'est le cas de *Tattie Daniel* pour lequel le total est aussi de 20.

10.4.2 Les relations vues selon le hashing

Les relations manipulées par l'interpréteur sont stockées en mémoire par une technique de hashing. Pour ce faire, un tableau de taille préfixée est réservé tout au début de l'exécution. Quand un nœud doit être stockée, une fonction mathématique dite *fonction de hashing* détermine quel doit être son emplacement dans le tableau. Ce procédé se base sur le numéro de variable du nœud ainsi que sur ses fils. La première fonction de hashing que nous avons choisie est :

$$\alpha = \left[(218.959.117 \cdot (1973 \cdot v + \sum_{i=1}^{n} fils_i \cdot 2^i)) \; div \; 97 \right] \; mod \; (MAX\_NOEUD - 3) + 2$$

où v est le numéro de la variable associée au nœud et MAX_NOEUD est la taille du tableau de hashing réservé en mémoire vive.

10.5 Structure utilisée

La table de hashing fait l'obet de la déclaration suivante :

```
TNoeud = CP   [   Label : Integer ;
                  Mult : Integer ;
                  Fils : Array of Integers ;
              ]

typedef int TRelation ;

#define FALSE (TRelation)0
#define TRUE (TRelation)1

RelationTable : TNoeud[MAX_NOEUD] ;
```

Ceci modélise trois choses

- Label permet de savoir à quelle variable le nœud est associé ;
- Mult comptabilise le nombre d'utilisations du nœud, permettant ainsi le partage entre des arbres ;
- Fils qui est un tableau renseignant les adresses des nœuds fils dans le tableau de hashing.

Le type TNœud est celui qui constitue le tableau de hashing, tandis que TRelation est le type qui est l'interface entre les tables de hashing avec ses nœuds et les relations telles qu'elles sont vue dans l'algorithme. A la manière d'un code orienté objet, il existe deux instance spéciales des relations et divers services offerts :

- les deux instances particulières sont les relations totalement vraie et totalement fausse, notées TRUE et FALSE. Elles occupent toujours, par convention, les deux premières cases du tableau de hashing car elles sont référencées par les entier 1 et 0. Ainsi, quand un nœud à un fils 1 ou 0, il renseigne ce nœud spécial, comme on a pu le voir sur diverses figures de relations avec partage de nœuds.
- l'observateur **GetLabel** : TRelation → Integer.
  v=GetLabel(r)
  pré : r est un entier désignant une relation dans le tableau de hashing
  post : v est le label de la variable associée à ce nœuds.

- l'observateur **GetFils** : TRelation → TRelation[ ]
  f=GetFils(r)
  pré : r est un entier désignant une relation dans le tableau de hashing
  post : f est le tableau de fils relatif à ce nœuds.

- le service **InitRelationTable** : Integer →
  InitRelationTable(n)
  pré : n est un entier supérieur à 2 strictement
  post : une table de hashing a été initialisée et préparée pour recevoir au maximum n nœuds de

relations. Les instances TRUE et FALSE sont placées aux deux premières cases de ce tableau.

- le constructeur **CreateRelation** : Integer × TRelation[ ] → TRelation
  r'=CreateRelation(v,$fils$[ ])
  pré : v est un label (numéro) de variable, et $fils$[ ] est un tableau d'entiers référant des relations dans la table de Hashing RelationTable. Plus formellement, il faut aussi que $Length(Fils[\ ]) = GetCardVar(v) \wedge \forall i\ v < GetLabel(Fils[i])$
  post : le nœud caractérisé par le label v et les fils $fils$[ ] a été ajouté dans la table. S'il n'existait pas encore, il a été créé de toute pièce. S'il existait déjà, le compteur de partage de ce nœud a été incrémenté d'une unité.

- le constructeur **CopyRelation** : TRelation → TRelation
  CopyRelation(r)
  pré : r réfère une relation dans la table de hashing
  post : le compteur de partage de la relation a été incrémenté d'une unité. Son but est d'éviter qu'un même objet ne soit détruit deux fois.

- le destructeur **DestroyRelation** : TRelation →
  DestroyRelation(r)
  pré : r réfère une relation dans la table de hashing
  post : le compteur de partage de la relation a été décrémenté d'une unité, et si ce nœud n'est plus utilisé (compteur de partage vaut 0) alors il est effacé de la table de hashing. Si la relation est $TRUE$ ou $FALSE$, le destructeur peut être appelé autant de fois que l'on veut car un tel appel n'a pas d'effet. Bien entendu, tout ceci est totalement transparent pour l'utilisateur.

- le service **PrintRelation** : Boolean →
  PrintRelation(b)
  pré : b est une valeur booléenne
  post : si le booléen est mis à faux, alors des informations succintes sur la table de hashing sont affichés. S'il est mis à vrai, alors en plus la table de hashing est affichée.

Ce dernier service manipule tout au long de l'exécution certaines variables comme **Relation_Table_Count**, **Max_Relation_Table_Count** dont le but est de connaître à tout moment le nombre de nœuds en cours d'utilisation ainsi que le plafond d'utilisation de la table. Ces variables interviennent dans la section 23.3 parlant des résulats de certains programmes. Enfin, ce service élabore d'autres valeurs statistiques comme le nombre de nœuds mal placés par rapport à l'adresse calculée par le fonction de hashing (l'adresse de base), ainsi que les distance moyennes et maximales séparant ces nœuds de leurs adresses de base.

Il faut garder une certaine prudence dans l'utilisation de ces services transparents. Il faut veiller à ce que chaque objet ne soit détruit qu'une et une seule fois. Considérons dans cette optique le petit bout de programme suivant :

```
function f(R : TRelation, S : TRelation) : TRelation;
    if R = TRUE
    then return S
    else return CreateRelation(...)

function toto
    ...
    S=f(R1,R2);
    DestroyRelation(R1);
    DestroyRelation(R2);
    DestroyRelation(S);
```

car il risque de poser des problèmes. En effet, si $R_1 = TRUE$, alors S et R_2 sont en fait le même objet. Les deux appels

DestroyRelation(R_2)

et

DestroyRelation(S)

vont détruire le même objet deux fois. La solution est de réécrire la fonction f comme suit :

```
...
    if R = TRUE
    then return CopyNoeud(S)
...
```

Avec l'appel au constructeur CopyNoeud, on aura toujours que R_2 et S sont des objets distincts (même s'ils peuvent être égaux comme des relations) et que leurs destructeurs peuvent être appelés indépendamment l'un de l'autre.

11 Les opérations relationnelles

module operations.c

interface :

CreateEgalite : Integer × Integer → TRelation

CreateInEgalite : Integer × Integer → TRelation

VarEgalConst : Integer × Integer → TRelation

CreateAppartenance : Integer × Integer list → TRelation

Et : TRelation × TRelation → TRelation

Ou : TRelation × TRelation → TRelation

Not : TRelation → TRelation

Diff : TRelation × TRelation → : TRelation

IlExiste : TRelation × Integer list → : TRelation

PourTout : TRelation × Integer list → : TRelation

TSubst = CP[
Var : Integer,
T : ENUM[SVar,SConst],
Value : Integer
]

Substituer : TRelation × TSubst list → : TRelation

utilise :

tliste.c
trelation.c

Les nœuds des relations sont placés dans une table de hashing appelée `RelationTable`. Notre implémentation utilise les opérations suivantes sur les relations :

1. Intersection de deux relations ;
2. Union de deux relations ;
3. Négation d'une relation ;
4. Différence entre deux relations ;
5. Projection d'une relation ;
6. Quantification existentielle sur une relation ;
7. Quantification universelle sur une relation ;

Les fonctions permettant ces opérations sont récursives.

11.1 Intersection de deux relations

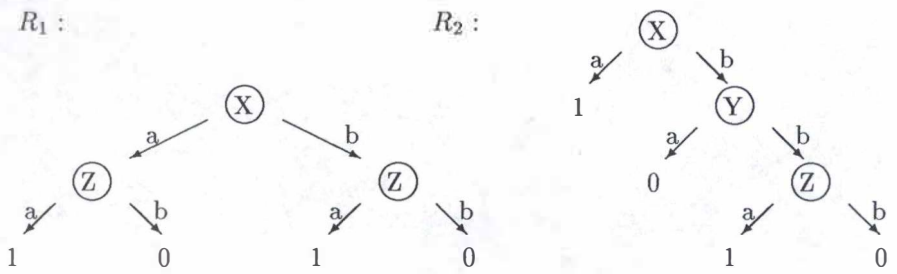
Etant données deux relations R_1 et R_2, il s'agit de calculer $R_1 \bigcap R_2$. Le pseudo-code de cette opération se présente comme suit :

```
fonction ET(r1, r2 : TRelation) : TRelation
    if (r1=false or r2=false) then return false
    if (r1=true) then return r2
    if (r2=true) then return r1
    x1←GetLabel(r1) ; x2←-GetLabel(r2) ; L←[ ] ; nd1←-GetFils(r1) ; nd2←-GetFils(r2)
    Case
        x1<x2 :   for i←0 to GetCardVar(x1)-1
                  do Add(L,ET(nd1[i],r2))
                  return CreateRelation(x1,L)
        x1>x2 :   for i←0 to GetCardVar(x2)-1
                  do Add(L,ET(r1,nd2[i]))
                  return CreateRelation(x2,L)
        x1=x2 :   for i←0 to GetCardVar(x1)-1
                  do Add(L,ET(nd1[i],nd2[i]))
                  return CreateRelation(x1,L)
```

Deux remarques doivent être faites :

- l'intruction Case qui permet de tester aisément les étiquettes des variables mises dans les nœuds de l'arbre, et ce en examinant successivement trois cas. Cela met en évidence le fait que ces étiquettes permettent un parcours descendant de l'arbre, tout en évitant des recherches inutiles et fastidieuses dans les différents niveaux de profondeur. Si c'eut été malheureusement le cas, une séquence de trois test n'aurait pas suffi.
- la présence de la fonction CreateRelation(label, liste_de_fils). Elle constitue à ce niveau une boite noire permettant de construire un nœud avec ses fils et de le placer dans le tableau `RelationTable`.

L'idée sous-jacente à cet algorithme est relativement simple. Considérons les deux relations suivantes :



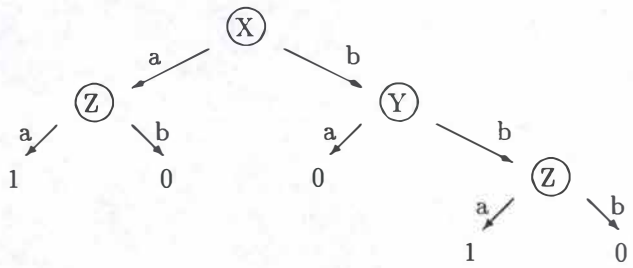
Le lecteur remarquera d'abord que les arbres représentés ici ne respectent pas les conventions d'optimisations développées plus tôt. En effet, seule l'omission des variables a été utilisée. Dans les exemples portant sur les opérations décrites dans ce paragraphe, nous ne respecterons que cette représentation car plus de zèle ne ferait qu'embrouiller le lecteur. De plus, le partage total des nœuds peut être découplé des opérations. Autrement dit, deux niveaux existent ici :

- le niveau *Méta* que nous examinons actuellement. Il est algorithmique, lié à une rédaction en pseudo-code ;
- le niveau le plus bas qui renferme la fonction CreateRelation(label, liste_de_fils). C'est elle qui prend en charge le placement correct des nœuds dans la table de hashing. Elle peut dès lors être accepetée comme boîte noire. Ceci justifie par ailleurs que les arbres sont représentés de façon moins pertinente

L'idée de ce pseudo-code est de procéder à l'intersection des fils des nœuds étiquetés par la même variable et de regrouper cette intersection sous le même label, c'est-à-dire la même étiquette de variable. La difficulté réside dans la convention supplémentaire d'omission de variables, ce qui implique, lorsque l'on avance plus profondément, un ajustement entre les labels. On conçoit alors tout l'avantage d'avoir placé les labels en ordre croissant selon la profondeur. Cet ajustement justifie les différentes possibilités d'appels récursifs présents dans l'instruction Case.

Plus concrètement, l'exploration commence par les branches de gauche, à partir de X. Etant donné que dans R_2, cette branche vaut ***vrai***, il suffira de recopier la branche gauche de R_1 comme étant un fils de "X". En avançant dans la branche de droite, il faut ajuster les étiquettes des variables car on a "Z" (resp. "Y") pour la relation R_1 (resp. R_2). L'omission de la variable "Y" dans R_1 signifiant que "Y" peut prendre à cet endroit n'importe quelle valeur, il suffit de recopier la structure de fils que possède "Y" dans la relation R_2 en intersectant le "Z" de R_1 avec les fils de "Y" (dans R_2)

Le résultat devient alors le suivant :

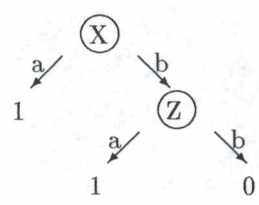


11.2 Union de deux relations

L'union de deux relations est du même tonneau. Tout ce qui, auparavant, était pensé en termes de conjonction doit être maintenant pensé en termes de disjonction. Le pseudo-code, très similaire au pseudo-code de l'intersection, est le suivant :

```
fonction OU(r1, r2 : TRelation) : TRelation
    if (r1=true or r2=true) then return true
    if (r1=false) then return r2
    if (r2=false) then return r1
    x1←GetLabel(r1) ; x2←GetLabel(r2) ; L←-[ ] ; nd1←GetFils(r1) ; nd2←GetFils(r1)
    Case
        x1<x2 :   for i←0 to GetCardVar(x1)-1
                  do Add(L,OU(nd1[i],r2))
                  return CreateRelation(x1,L)
        x1>x2 :   for i←0 to GetCardVar(x2)-1
                  do Add(L,OU(r1,nd2[i]))
                  return CreateRelation(x2,L)
        x1=x2 :   for i←0 to GetCardVar(x1)-1
                  do Add(L,OU(nd1[i],nd2[i]))
                  return CreateRelation(x1,L)
```

Si nous reprenons les deux relations de la page 76, nous trouvons alors



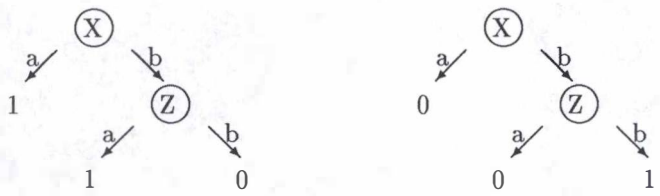
11.3 Négation d'une relation

Par *Négation d'une relation*, on entend la simple manipulation de l'arbre où l'on inverse les rôles des valeurs 1 et 0. Il ne faut donc pas voir dans ce paragraphe la seconde composante d'une birelation que nous avions appelée *Neg*. La négation que nous examinons ici se situe à un autre niveau que la négation d'un prédicat logique de notre langage.
Le pseudo-code est très rudimentaire :

```
fonction NOT(r : TRelation) : TRelation ;
    if r=true then return false
    if r=false then return true
    L←-[ ] ; nd←-GetFils(r) ;
    for i←-0 to GetCardVar[GetLabel(r)]-1
    do Add(L,NOT(nd[i]))
    return CreateRelation(r,L)
```

On illustre ce cas par un exemple, sans que cela nécessite plus d'explication.

R devient *not R*



11.4 Différences entre deux relations

Il s'agit d'une manipulation au sens ensembliste :

$$A \setminus B = A \cap B^c$$

Le pseudo-code est :

```
fonction DIFF(r1, r2 : TRelation) : TRelation
    r2←-NOT(r2)
    return ET(r1,r2)
```

11.5 Projection d'une relation

Certaines opérations sont plus élaborées que l'intersection ou la réunion ensemblistes de deux ensembles. C'est le cas de la *projection*. Pour ce faire, deux éléments doivent être donnés :

- la relation projetée ;
- l'ensemble des variables sur lesquelles on projette.

Etant donné que les arbres sont construits en liant étroitement profondeur et étiquette de variable, on suppose que la liste des étiquettes de variable décrivant l'espace où l'on projette est triée par ordre croissant. Cette hypothèse évite des allées et venues inutiles entre les différentes profondeurs de l'arbre.
Le pseudo-code de cette partie commence à se complexifier :

```
fonction Projection(r : TRelation, VarListe : SEQ[Var]) : TRelation
   if (Length(VarListe)=0)
   then    if (r=false)
           then return false;
           else return true;
   else    if (r=true or r=false)
           then return r;
           x←GetLabel(r) ; nd←GetFils(r) ; M←[ ];
           Case
              VarListe[1]=x :
                 for i←0 to GetCardVar(x)-1
                 do Add(M,Projection(nd[i],VarListe[2..length(VarListe)])
                 return CreateRelation(x,M) ;
              VarListe[1] < x :
                 return Projection(r,VarListe[2..length(VarListe)])
              VarListe[1] > x :
                 return Projection(Disjonction_de_Noeuds(nd),VarListe;
```

où Disjonction_de_Noeuds(liste_de_noeuds) renvoie le résultat d'une opération de disjonction portant sur tous les sous-arbres dans liste_de_noeuds. Dans le code, cette fonction Disjonction_de_Noeuds ne figure pas telle quelle mais sous la forme d'une boucle. Enfin, il faut quelques commentaires supplémentaires.

Les relations totalement vraies ou totalement fausses Ces relations, peu importe l'espace sur lequel on projette, donnent toujours elles-mêmes.

Les autres relations Selon la même philosophie que pour la conjonction ou la disjonction, on envisage différents cas l'un après l'autre. Dans ces opérations, on comparait les étiquettes des nœuds de l'arbre afin de progresser de la même façon dans les deux arbres. Pour la projection, on procède similairement à un ajustement entre les étiquettes des nœuds de l'arbre et les étiquettes de l'espace. Ceci explique la séquence des trois tests.

11.6 Quantification existentielle sur une relation

En assimilant une relation à la structure d'arbre que nous avons définie pour la représenter, nous pouvons réécrire un prédicat en fonction des fils du nœud racine de l'arbre. Soit $p(X_1, \ldots, X_n)$ un prédicat, et $\{a_1, \ldots, a_m\}$ le domaine de X_1. Lorsque l'on rencontre un quantificateur existentiel, on peut remarquer que :

$$
\begin{aligned}
&\exists X_1\ p(X_1, \ldots, X_n) \\
&\Leftrightarrow \exists X_1\ \ [X_1 = a_1\ \wedge q_1(X_2, \ldots, X_n)] \vee \ldots \vee [X_1 = a_m\ \wedge q_m(X_2, \ldots, X_n)] \\
&\Leftrightarrow \exists X_1\ [X_1 = a_1\ \wedge q_1(X_2, \ldots, X_n)] \vee \ldots \vee\ \exists X_1\ [X_1 = a_m\ \wedge q_m(X_2, \ldots, X_n)]
\end{aligned}
$$

Ceci montre clairement qu'il suffit de procéder à une disjonction des fils de X_1. De plus, on gardera toujours à l'esprit la notion d'ajustement dont il a été souvent question jusqu'ici. Ces deux principes régissent le pseudo-code ci-dessous :

```
fonction IlExiste(r :TRelation ; s : SEQ[TVar]) : TRelation ;
    if s=[ ]
    then    return r
    else    x←GetLabel(r) ; nd←GetFils(r) ; L←[ ]
            Case
                s[0] < x :    return IlExiste(r,s[1..length(s)-1])

                s[0] > x :    for i←0 to GetCardVar(x)-1
                              do Add(L,IlExiste(nd[i],s))
                              return CreateRelation(x,L)
                s[0] = x :    for i←0 to GetCardVar(x)-1
                              do Add(L,IlExiste(nd[i],s[1..length(s)-1]))
                              return Disjonction_de_Noeud(L)
```

11.7 Quantification universelle sur une relation

Le lecteur aura compris que le principe de *quantification universelle* ne diffère pas énormément du principe de *quantification existentiel*. Le pseudo-code a juste subi quelques retouches, transformant disjonctions en conjonctions :

```
fonction PourTout(r :TRelation ; s : SEQ[TVar]) : TRelation ;
    if s=[ ]
    then    return r
    else    x←GetLabel(r) ; nd←GetFils(r) ; L←[ ]
            Case
                s[0] < x :    return PourTout(r,s[1..length(s)-1])

                s[0] > x :    for i←0 to GetCardVar(x)-1
                              do Add(L,PourTout(nd[i],s))
                              return CreateRelation(x,L)
                s[0] = x :    for i←0 to GetCardVar(x)-1
                              do Add(L,PourTout(nd[i],s[1..length(s)-1]))
                              return Conjonction_de_Noeud(L)
```

On remarquera ici l'existence de la fonction Conjonction_de_Noeuds(liste_de_noeuds) dont le rôle est dual de celui de Disjonction_de_Noeuds(liste_de_noeuds).

11.8 Les substitutions

Pour implémenter l'opération de substitution que nous avons spécifiées au paragraphe (2.3.8), nous utiliserons le type suivant :

```
TSubst = CP[
            Var : Integer,
            T : ENUM[SVar,SConst],
            Value : Integer,
            ]
```

Une substitution dans le sens mathématique peut alors être codée comme une liste de *TSubst*.

Par exemple, la substitution

$[X \to Y, Z \to a]$,

deviendrait

$[<\text{Var} : X, T : \text{SVar}, \text{Value} : Y >; <\text{Var} : Z, T : \text{SConst}, \text{Value} : a >]$.

La fonction

Substituer : TRelation × TSubst list → TRelation,

permet alors d'effectuer la substitution proprement dite.

Au niveau de l'algorithme cette substitution est réalisée en deux phases :

- Dans une première phase, l'arborescence pour la relation d'entrée est parcourue une première fois. Si on rencontre un nœud étiqueté par X et si dans la liste des substitutions, il est indiqué que l'étiquette X doit être remplacée par Y, ce renommage est effectué.
  De la même façon, si dans la liste des substitutions, on trouve que l'étiquette X doit être remplacée par la constante c, on remplace le nœud en question par le nœud fils qui correspond à cette constante.

  En pseudo-code, cette opération est réalisée par la fonction suivante :

```
function SubstituerLabels(R : TRelation, Subst : TSubst list) : TRelation
begin
    Fils ←GetFils(R);
    X ←GetLabel(R);
    S ←nil;
    for i = 0 to Subst.Count do
        if Subst[i].Var = X then S ←Subst[i];
    if S ≠ nil
    then if S.T = SConst
        then return SubstituerLabels(Fils[S.Value],Subst);
        else NouveauLabel ←S.Value;
    else NouveauLabel ←X;
    for i = 0 to GetCardVar(X)
        do NouveauFils[i] ←SubstituerLabels(Fils[i],Subst);
    return CreateRelation(NouveauLabel,NouveauFils);
end
```

- Après que la fonction *SubstituerLabels* ait remplacé les étiquettes des nœuds par des nouvelles, une restructuration de l'arborescence obtenue peut être nécessaire pour assurer que l'étiquette de chaque nœud est inférieure aux étiquettes de ces fils.
  La fonction *SubstLabels* peut en effet avoir violé cette contrainte si la substitution d'entrée est non-monotone, dans le sens où elle contient deux termes

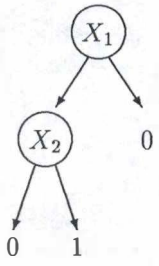
  $$X \to X' \text{ et } Y \to Y'$$

  où $X < Y$, mais $X' \geq Y'$.

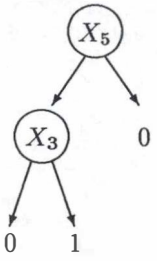
  Ainsi, si on considère la substitution

  $$[X_1 \to X_5, X_2 \to X_3],$$

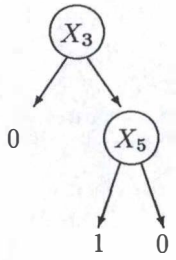
  pour la relation :



  l'arborescence obtenue par une substitution aveugle des étiquettes :



ne peut être considérée comme une représentation valide d'une relation, que si on la transforme en l'arborescence :



L'algorithme utilisé pour cette réorganistion est assez simple : pour réorganiser un nœud, on réorganise d'abord ses fils et après on utilise les opération *Et* et *Ou* pour les remettre ensemble.

Ceci est décrit par le pseudo-code ci-dessous :

```
function Reorganiser(R : TRelation) : TRelation
begin
    if (R = TRUE or R = FALSE) then return TRUE;
    Out ←-FALSE;
    X ←-GetLabel(R);
    for i = 0 to GetCardVar(X)-1 do
        Fils ←-Reorganiser(GetFils(R)[i]);
        Out ←-Ou(Out, Et(VarEgalConst(X,i),Fils));
    return Out;
end;
```

La function *Substituer* n'a alors qu'a assurer l'enchaînement de ces deux phases :

```
function Substituer(R : TRelation,Subst : TSubst list)
begin
    Out = SubstituerLabels(R,Subst);
    if ContientNonMonotonie(Subst)
        then Out ←-Reorganiser(Out);
    return Out;
end;
```

Remarquons encore que, comparée aux autres opérations relationelles, cette opération de substitution est assez coûteuse dans le cas où l'on est confronté à des non-monotonies.

En effet, toutes les autres opérations ne nécessite pour leur réalisation qu'un seul parcours de chaque arborescence d'entrée. On voit facilement que ceci est loin d'être le cas pour la fonction de restructuration.

12 Les types syntaxiques

Dans ce chapitre nous présenterons les types syntaxiques.
On remarque que ces types seront utilisés lors des trois phases (parsage, calcul et impression) du logiciel. Ces types doivent par conséquence permettre de représenter non seulement la syntaxe abstraite, mais doivent aussi garder une trace de la plupart des éléments de la syntaxe concrète.
Nous présenterons les types utilisés à cet égard d'une façon *top-down* pour qu'on voit mieux comment ils s'articulent.
Pour les définitions des domaines on utilise les types suivants :

- Un domaine est caractérisé par un nom, une description et le nombre d'éléments qu'il contient.

```
TDomaine = CP[
                Nom : String,
                Definition : TDescrDomaine,
                Count : Integer,
                ]
```

- Une description de domaine est soit la description d'un domaine énuméré, soit la description d'un domaine intervalle.

```
TDescrDomaine = UNION[TDomEnumere, TDomIntervalle]
```

- Un domaine énuméré est décrit par une liste de noms de constantes.

```
TDomEnumere = SEQ[String]
```

- Un domaine intervalle est décrit par un nom, une borne inférieure et une borne supérieure.

```
TDomIntervalle = CP[
                    Nom : Integer,
                    BInf : Integer,
                    BSup : Integer,
                    ]
```

Pour les prédicats on utilise les types suivants :

- Un prédicat est caractérisé par
  - un nom,
  - une liste de conjonctions quantifiées,
  - une liste de paramètres typés,
  - la liste des numéros des variables utilisées dans cette liste de paramètres,
  - une deuxième liste de numéros de variables, obtenue par le tri de la liste précédente,
  - deux relations qui représenterons les ensembles des tuples où on a trouvé que le prédicat est vrai, respectivement faux.

```
TPredicat = CP[
                Nom : String,
                Conjonctions : SEQ[TConjonction],
                ParamTypes : SEQ[TParamType],
                Vars : SEQ[Integer],
                VarsSorted : SEQ[Integer],
                Pos : TRelation,
                Neg : TRelation
                ]
```

- Un paramètre typé est caractérisé par un nom de variable, un nom de domaine et le numéro de ce même domaine.

```
TParamType = CP[
                NomVar : String,
                NomDomaine : String,
                Domaine : Integer
                ]
```

- Une conjonction (quantifiée) est caractérisée par une séquence de contraintes, une séquence de littéraux et une séquence contenant les numéros des variables quantifiées.

```
TConjonction = CP[
                Contraintes : SEQ[TContrainte],
                Literaux : SEQ[TLiteral],
                VarsQuants : SEQ[Integer],
                ]
```

- Une contrainte est soit une comparaison de variables, soit une appartenance.

```
TContrainte = UNION[TComparVar, TAppartenance]
```

- Une comparaison de variables est caractérisée par deux noms de variables, les numéros de variables correspondants, l'opérateur qui les met en relation et le numéro de la ligne où la comparaison apparaît dans le programme.

```
TComparVar = CP[
                NomVar1 : String,
                Var1 : Integer,
                Op : TOperateur,
                NomVar2 : String,
                Var2 : Integer,
                NLigne : Integer
                ]
```

- Il y a deux opérations de comparaison : l'égalité et l'inégalité :

```
TOperateur = ENUM[EQ,NE]
```

- Une appartenance est caractérisée par un nom de variable et le numéro correspondant, un numéro de domaine, une description d'un sous-domaine, un ensemble d'entiers qui contient les numéros des constantes dans ce sous-domaine et le numéro de la ligne où l'appartenance apparaît dans le programme.

```
TAppartenance = CP[
                NomVar : String,
                Var : Integer,
                Domaine : Integer,
                Ensemble : TDescrDomaine,
                Constantes : SEQ[Integer],
                NLigne : Integer
                ]
```

- Un littéral est caractérisé par un signe et un appel.

```
TLiteral = CP[Signe : TSigne, Appel : TAppel]
```

- Un signe est soit positif, soit négatif.

TSigne = ENUM[POS,NEG]

- Un appel est caractérisé par un nom de prédicat et le numéro de prédicat correspondant, une liste de paramètres effectifs et le numéro de ligne ou l'appel apparaît dans le programme.

```
TAppel = CP[
            NomPred : String,
            Pred : Integer,
            ParamsEffs : SEQ[TVarOuConst],
            NLigne : Integer,
            ]
```

- Un paramètre effectif est soit une variable, soit une constante.

TVarOuConst = UNION[TVar, TConst]

- Une variable est caractérisée par un nom et un numéro.

```
TVar = CP[
          NomVar : String,
          Var : Integer
          ]
```

- Une constante est caractérisée par une description, le numéro du domaine qui la contient et son rang dans ce domaine.

```
TConst = CP[
            Def : TDescrConst,
            Dom : Integer
            Const : Integer
            ]
```

- La description d'une constante est la description soit d'une constante élémentaire, soit une constante indexée.

TDescrConst = UNION[TConstElem, TConstIndexee]

- La description d'une constante élémentaire est son nom.

TConstElem = String

- La description d'une constante indexée est caractérisée par le nom d'une intervalle et un index.

TConstIndexee = CP[Nom : String, Index : Integer]

13 Les tables syntaxiques

module tables.c

interface :

InitTables : →

PredTable : SEQ[TPredicat]

GetNrPredicat : String → TPredicat

GetPredicat : Integer → TPredicat

DomTable : SEQ[TDomaine]

GetDomaine : Integer → TDomaine

StringOfConst : Integer × Integer → String

NVariables : Integer

NomVarTable : SEQ[String]

DomVarTable : String[ ]

GetNDomVar : Integer → Integer

GetDomVar : Integer → TDomaine

CardVarTable : Integer[ ]

GetCardVar : Integer → Integer

utilise :

tliste.c

Ce module exporte les tables dans lesquelles seront stockées toutes les informations syntaxiques extraites du programme à interpréter.

Concrètement on a les variables globales suivantes :

- **PredTable** : SEQ[TPredicat]
  où seront stockés les prédicats.
  On remarquera que, quand on parle du numéro d'un prédicat, on fait en fait référence à l'index du prédicat dans cette table.
- **DomTable** : SEQ[TDomaine]
  où seront stockés les domaines.
  On remarquera que quand on parle du numéro d'un domaine, on fait en fait référence à l'index du domaine dans cette table.
- **NVariables** : Integer
  Le nombre des variables utilisées dans le programme à interpréter.
  Les numéros de ces variables sont donc $0 \ldots NVariables - 1$.
- **NomVarTable** : SEQ[String]
  Cette table permet de récupérer le nom d'une variable dont on connaît le numéro.

- **DomVarTable** : Integer[ ]
  Ce tableau (de longueur *NVariables*) permet de récupérer le numéro du domaine auquel appartient une variable.
- **CardVarTable** : Integer[ ]
  Ce tableau (de longueur *NVariables*) permet de récupérer la cardinalité du domaine auquel appartient une variable.

On remarquera que l'on n'a pas vraiment besoin d'un tableau séparé pour récupérer la cardinalité d'une variable, les tables **DomVarTable** et **DomTable** suffisent à cette fin.
Nous avons introduit le tableau **CardVarTable** par un souci d'efficacité. En effet, la cardinalité d'une variable est une information régulièrement consultée par les fonctions qui implémentent les opérations relationnelles. Comme c'est principalement de ces fonctions que dépend le temps de réponse de notre logiciel, il y a tout intérêt à optimiser cet accès autant que possible.
C'est pour cette même raison que nous avons choisi d'implémenter *CardVarTable* comme un tableau et pas comme une liste. Comme ça, chaque accès se fait pas une simple consultation d'un tableau et pas par un appel de fonction.

Le module *tables.c* exporte aussi une fonction **InitTables** qui permet d'intitialiser toutes les tables.

Dans le reste du programme, les accès aux informations syntaxiques se font principalement par des accès aux tables ci-dessus. Néanmoins ce module offre quelques fonctions pour faciliter les requêtes les plus courantes ou moins évidentes. Concrètement, on a implémenté les fonctions ci-dessous :

- **GetNrPredicat** : String → Integer
  i=GetNrPredicat(nom)
  post: si un prédicat avec le nom *nom* existe, i est le numéro de ce prédicat, sinon i est -1.
- **GetPredicat** : Integer → TPredicat
  P=GetPredicat(i)
  pré : $0 \leq i <$ Count(PredTable)
  post: P est le prédicat dont le numéro est i.
- **GetDomaine** : Integer → TDomaine
  D=GetDomaine(i)
  pré : $0 \leq i <$ Count(DomTable)
  post: D est le domaine dont le numéro est i.
- **StringOfConst** : Integer × Integer → String
  s=StringOfConst(dom,const)
  pré : $0 \leq$ dom $<$ Count(DomTable)
  $0 \leq$ const $<$ GetDomaine(dom).Count
  post: s est la description textuelle de la *const*-ième constante du domaine dont le numéro est *dom*.
- **GetNDomVar** : Integer → Integer
  dom=GetNDomVar(v)
  pré : $0 \leq v <$ NVariables
  post: *dom* et le numéro du domaine auquel appartient la variable avec le numéro v.
- **GetDomVar** : Integer → TDomaine
  dom=GetDomVar(v)
  pré : $0 \leq v <$ NVariables
  post: *dom* et le domaine auquel appartient la variable avec le numéro v.
- **GetCardVar** : Integer → Integer
  card=GetNDomVar(v)
  pré : $0 \leq v <$ NVariables
  post: *card* est la cardinalité du domaine auquel appartient la variable avec le numéro v.

  Comme cette fonction est très fréquemment appelée par les fonctions qui implémentent les opérations relationnelles, on a intérêt à implémenter cette fonction comme une fonction *inline*.

14 Les tokens du β-langage

module scan.l
interface : **yyin** : FILE **yytext** : $\rightarrow$
utilise : $\emptyset$

Ce module caractérise une seule fonction **yytext** utilisée par **yyparse** (cfr plus loin) et par elle seule. Son rôle est de reconnaître les mots constituant le fichier source et d'en avertir au fur et à mesure la fonction **yyparse** de plus haut niveau (syntaxique et non plus lexical). Les mots sont

- des mots réservés comme domain, in, not ;
- élimination des commentaires du programmeur qui sont non pertinents ou injurieux, des blancs et des retours chariots ;
- les identificateurs consitutés des caractères a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, _, ainsi que les entiers ;
- circonscription de symboles précis : ~, .. , :=, =, <>, et le point-virgule.

15 Le parsage

module parse.y
interface : **yyin** : FILE **yyparse** : →
utilise : tables.c tliste.c

Ce module exporte une seule fonction **yyparse** qui effectue une première analyse syntaxique du programme source qui se trouve dans le fichier **yyin**.

Cette fonction vérifie si les règles de production de la syntaxe concrète (cfr. chapitre (5)) sont bien respectées et vérifie aussi quelques contraintes syntaxiques de plus haut niveau. En faisant cela, elle peut renvoyer les messages d'erreur 3 à 14 (cfr. paragraphe (5.4)).

Si aucune erreur syntaxique n'est découverte, la structure syntaxique sera codée en utilisant les types syntaxiques introduits au chapitre (12).

Il importe de remarquer que dans ces types syntaxiques, tous les attributs qui ont été mis en italique lors de la présentation, ne peuvent pas encore être déterminés sur ce niveau-ci.
Ces attributs seront initialisés selon les règles suivantes :

- un entier est initialisé à -1
- un booléen à *false*
- une relation à *FALSE*
- une liste à la liste vide

Chaque *TDomaine* (resp. *TPredicat*) construit est ajouté à la table *PredTables* (resp. *DomTable*).

Les autres tables syntaxiques ne sont pas utilisées par ce module.

16 Le numerotage

module numerotage.c
interface : **NumeroterPredicats** : → **NumeroterVars** : →
utilise : tables.c tliste.c

Ce module exporte deux fonctions.

La première, **NumeroterPredicats**, va parcourir toutes les définitions de prédicats dans *PredTable* et initialiser tous les attributs *TAppel.Pred*.

En faisant ceci, elle peut se rendre compte qu'un prédicat appelé n'est pas déclaré dans le programme à analyser ou bien, qu'il est déclaré, mais avec une arité différente. Si cette situation se produit, un message d'erreur de type 15 ou 16 (cfr. paragraphe (5.4)) est renvoyé et l'interpréteur est interrompu.

La deuxième fonction **NumeroterVariables** est plus complexe. Elle va affecter un numéro à chaque variable du programme et initialiser tous les attributs dans les types syntaxiques destinés à contenir ces numéros de variable (c'est-à-dire, tous les attributs qui ont été mis en italique lors de la présentation des types syntaxiques au chapitre (12)).

Au niveau des tables syntaxiques, la fonction va initialiser la variable *NVariables* et la table *NomVarTable* qui permettra de récupérer le nom que l'utilisateur a donné à une variable.

La raison principale pour laquelle nous avons choisi de numéroter les variables et d'utiliser ces numéros plutôt que les noms choisis par l'utilisateur, est que ces derniers ne sont pas des identifiants pour les variables.

Ainsi dans l'exemple

$$\begin{cases} \text{domain}\{a, b\}; \\ \text{domain dom}\{p, q, r\}; \\ p(X) \ := \ X = Y \ \& \ Y = a \ \mid \ X = b \ \& \ Y = p; \end{cases}$$

il y a en fait deux variables qui portent le nom Y. Et, ce qui est encore plus gênant, c'est que ces deux variables n'ont pas le même type, ni même la même cardinalité !

Pourtant, il est intéressant de disposer d'un moyen efficace pour déterminer la cardinalité d'une variable donnée.

Si on numérote les variables, ceci peut être fait par un simple accès à une table

VarCardTable,

réalisé par une fonction (programmé *inline* pour des raisons d'efficacité)

GetCardVar : Integer → Integer.

On aura remarqué que cette fonction est cruciale pour tout ce qui est de notre implémentation des relations, où elle permet de déterminer le nombre de fils d'un nœud étiqueté par une variable donnée.

L'alternative aurait été d'ajouter dans chaque nœud un attribut contenant son nombre de fils, mais comme le nombre de nœuds dans une application d'une certaine taille, s'élève facilement à quelques

dizaines de milliers, ceci n'est pas une bonne approche du point de vue consommation mémoire.

Pour ce qui est de l'algorithme de numérotage proprement dit, nous nous sommes, dans cette première version du logiciel, limités à implémenter un algorithme tout simple qui considère chaque définition de prédicat indépendamment et qui attribue des numéros aux variables au fur et à mesure qu'elles apparaissent dans ces définitions.
Ainsi pour le programme

$$\begin{cases} p(X,Y) & := \quad X <> Z \ \& \ q(X,Z) \ \mid \ Z \text{ in } \{a,b\} \ \& \ q(Y,Z); \\ q(X,Y) & := \quad X = a \ \& \ Y = b; \end{cases}$$

on obtiendra le numérotage ci-dessous :

$$\begin{cases} p(X_0,X_1) & := \quad X_0 <> X_2 \ \& \ q(X_0,X_2) \ \mid \ X_3 \text{ in } \{a,b\} \ \& \ q(X_1,X_3); \\ q(X_4,X_5) & := \quad X_4 = a \ \& \ X_5 = b; \end{cases}$$

Dans de futures versions du logiciel, on devra absolument essayer de développer un meilleur algorithme. En effet, on s'est rendu compte que le numérotage effectué a un impact énorme sur le temps de réponse et la consommation mémoire de notre logiciel.

Il y a essentiellement deux raisons pour cela :

- La raison principale semble être que le numérotage effectué, circonscrit d'une certaine manière la possibilité de faire un partage des nœuds.
  En effet, deux nœuds ne peuvent être partagés que s'ils sont identiques, c'est-à-dire, s'ils ont les mêmes fils *et* la même étiquette.
  Ainsi, dans l'exemple ci-dessus, *aucun* partage des nœuds n'est possible entre les relations utilisées pour les calculs de p et de q, puisque les étiquettes utilisées dans leurs définitions, sont deux à deux distinctes.
  On voit donc que l'algorithme de numérotage très primitif que nous avons implémenté, empêche à un certain point au mécanisme du partage des nœuds de réaliser toute l'économie mémoire possible.
  Dans l'exemple considéré, un numérotage tel que

$$\begin{cases} p(X_0,X_1) & := \quad X_0 <> X_2 \ \& \ q(X_0,X_2) \ \mid \ X_2 \text{ in } \{a,b\} \ \& \ q(X_1,X_2); \\ q(X_0,X_2) & := \quad X_0 = a \ \& \ X_2 = b; \end{cases}$$

  ferait beaucoup mieux sur ce point.
- Une deuxième raison est que, comme nous le verrons au paragraphe (18.3.5), l'évaluation d'un appel de prédicat par notre interpréteur nécessite une substitution de variables dans les relation associées au prédicat appelé.

  Nous avons fait remarquer au paragraphe (11.8), qu'une telle substitution peut être une opération très efficace ou très lourde, selon que des non-monotonies sont présentes dans cette substitution ou non.

  Un algorithme de numérotage qui permettrait de minimiser le nombre de non-monotonies dans les appels de prédicat pourrait avoir un effet favorable sur le temps de réponse de notre logiciel.

Remarquons encore que, pour autant que deux variables avec des cardinalités différentes ne reçoivent pas le même numéro, les autres modules dans notre architecture sont complètement indépendants de l'algoritme de numérotage choisi. [12]

Ceci devrait faciliter considérablement la mise en œuvre d'une nouvelle version.

[12] La seule exception est le module *printnoeud* qui, dans la version actuelle, accède à la table *NomVarTable* pour déterminer le nom d'une variable à partir de son numéro.
Si l' on veut permettre que l'algorithme de numérotage attribue le même numéro à deux variables portant des noms différents, ce module nécessitera quelques petites modifications

17 Le typage

module typage.c
interface : **Typage : →**
utilise : tables.c tliste.c

Ce module exporte une seule fonction **Typage** qui va essayer de déterminer le type de chaque variable et de vérifier si tout le programme est libre de conflits de type.

Si un conflit de type est détecté ou si le type d'une variable ne peut être déterminé, un message d'erreur de type 18 ou 19 est renvoyé.

Sinon, la fonction va initialiser les tables syntaxiques *DomVarTable* et *CardVarTable* qui permettront plus tard d'accéder rapidement au domaine d'une variable donnée, ainsi qu'à la cardinalité de cette variable.

18 La résolution d'une requête

module pointfixe.c

interface :

CalculerPointFixe : ESigne × String → TRelation
Out = CalculerPointFixe(Signe,NomPredicat)

utilise :

operations.c
trelation.c
tables.c
tliste.c

18.1 Introduction

Etant donné un programme P, nous devons pouvoir répondre aux requêtes sur P comme spécifié au paragraphe (7.5).
Il s'agît pour cela de combiner ce que nous avons vu aux chapitres (7) et (4).

Rappelons brièvement ce dont nous disposons :

- Etant donné un ensemble A, un ensemble A' et un opérateur continu
  $$\tau : (A \to A') \to (A \to A'),$$
  l'algorithme développé au chapître (4) permet de calculer $(\mu\ \tau)(a)$, la valeur que prend le point fixe de τ dans un élément $a \in A$ donné.
  Pour pouvoir utiliser cet algorithme on a seulement besoin d'une fonction effective
  $$\text{tau} : (A \to A') \times A \to A'$$
  telle que $(\tau\ f)(a) = tau(f, a)$.
- Nous avons défini la sémantique d'un programme comme le point fixe d'un opérateur continu
  $$\text{XB}_P : \Omega_P \to \Omega_P$$
  où Ω_P, l'ensemble des interprétations de P, est un sous ensemble de l'espace des fonctions $\{p_0, \ldots, p_n\} \to \mathcal{SB}$ de l'ensemble des prédicats dans l'ensemble des birelations.
  Cet opérateur a été défini à l'aide d'un opérateur $\mathcal{F}_B$, tel que pour chaque $B \in \Omega_P$ et chaque prédicat p :
  $$(\text{XB}_P\ B)(p) = \mathcal{F}_B(D)$$
  où D est la disjonction dans la définition de p.

Avec tout ceci, on peut répondre aux requêtes sur P comme suit :
Dans l'algorithme de point fixe on prend comme

- ensemble A, l'ensemble $\{p_0, \ldots, p_n\}$ des prédicats,
- treillis complet A', l'ensemble $\mathcal{SB}$ des birelations
- opérateur τ, l'opérateur XB_P,
- fonction effective, la fonction
  $$\text{tau}(B, p) = \mathcal{F}_B(D)$$
  où D est la disjonction dans la définition de p.

En appliquant l'algorithme nous pouvons alors, pour un prédicat p donné, calculer la birelation $(\mu\ \text{XB}_P)(p)$.
La réponse à la requête $?p$ (resp. $?\neg p$) est alors la partie positive (resp. négative) de cette birelation.

18.2 Calculer les parties positives et négatives

Bien que nous puissions répondre aux requêtes comme indiqué au paragraphe précédent, cette manière de procéder n'est pas très efficace et ceci pour la raison suivante :

Si l'utilisateur fait une requête $?p$ ou $?\neg p$, l'algorithme proposé va calculer aussi bien la partie positive que la partie négative de la birelation qui correspond au prédicat p. Par contre, si la requête était $?p$ (resp. $?\neg p$), tout ce que demande l'utilisateur est la partie positive (resp. négative) de cette birelation et il est clair que l'algorithme risque de faire pas mal de calculs inutiles.

Ce problème est facile à résoudre, bien que l'on doive pour cela partir d'une formulation de la sémantique qui est légèrement différente.

Dans l'ancienne version, on a considéré une interprétation d'un programme comme étant une fonction qui associe à chaque prédicat une birelation.
A partir de maintenant, nous allons considérer qu'une interprétation associe à un signe et un prédicat une relation, où cette dernière contient les tuples où le prédicat est connu vrai ou faux selon le signe en entrée.

Soit Ω'_P l'ensemble de ces interprétations :

$$\Omega'_P = \{+,-\} \times \{p_0, \ldots, p_\nu\} \to \mathcal{SR}$$

On remarque que les deux ensembles Ω_P et Ω'_P sont strictement isomorphe. En effet, à chaque interprétation $B \in \Omega'_P$ correspond une interprétation $\widehat{B} \in \Omega_P$ par la transformation

$$\forall p : \widehat{B}\ p = \langle B(+,p), B(-,p) \rangle .$$

On peut alors définir un opérateur XB'_P, qui prend une approximation de la sémantique de P et en renvoie une nouvelle :

$$\mathrm{XB}'_P : \Omega'_P \to \Omega'_P.$$

Cette opérateur peut être défini à l'aide de l'opérateur XB_P que nous avions comme suit :

$$\big\langle (\mathrm{XB}'_P\ B)(+,p), (\mathrm{XB}'_P\ B)(-,p) \big\rangle = \mathrm{XB}_P\ \widehat{B}$$

pour chaque $B \in \Omega'_P$ et pour chaque prédicat p.

Comme on sait que l'opérateur XB_P est continu, on peut facilement démontrer que ceci vaut aussi pour l'opérateur XB'_P.

On peut alors définir la réponse à une requête

$$?\ p \text{ (resp. } ?\ \neg p)$$

comme

$$(\mu\ \mathrm{XB}'_P)(+,p) \text{ (resp. } (\mu\ \mathrm{XB}'_P)(+,p)).$$

Cette définition est complètement équivalente à la définition que l'on a donné au paragraphe (7.5) mais qui, cette fois-ci, permet une application directe de l'algorithme générique de calcul de point fixe qui considère indépendamment les parties positives et négatives des birelations.

Concrètement, on peut maintenant, dans l'algorithme de point fixe, prendre comme

- ensemble A, l'ensemble
  $$\{+,-\} \times \{p_0, \ldots, p_n\}$$
  des signes et prédicats,
- treillis complet A', l'ensemble $\mathcal{SR}$ des relations,
- opérateur τ, l'opérateur XB'_P,
- fonction effective, la fonction telle que pour chaque $B \in \Omega'_P$, chaque prédicat p et chaque signe s :
  $$tau(B, < s, p >) = \begin{cases} \mathcal{F}^+_B(D) & \text{si } s = + \\ \mathcal{F}^-_B(D) & \text{si } s = - \end{cases}$$
  où D est la disjonction dans le définition de p et où $\mathcal{F}^+_B(D)$ (resp. $\mathcal{F}^-_B(D)$) est la partie positive (resp. négative) de $\mathcal{F}_B(D)$.

18.3 Le pseudo-code

C'est essentiellement l'idée ci-dessus que nous avons implémentée, comme décrite dans le pseudo-code ci-dessous.

Les types utilisés sont, d'une part les types syntaxiques, introduits au chapitre (12) et d'autre part le type

```
TSommet=CP[
            Signe : TSigne;
            Predicat : TPredicat;
            Utilise : SET[TSommet];
            UtilisePar : SET[TSommet]
            ]
```

utilisé comme dans l'algorithme du point fixe générique.

Les variables globales utilisées sont les suivantes :

```
PredTable : SEQ[TPredicat]
Dg : SEQ[TSommet]
SommetAppelant : TSommet
```

où les deux dernières seront utilisées comme dans l'algorithme générique et la table des prédicats permettra, non seulement de repérer la définition d'un prédicat donné, mais aussi (par le biais des attributs *Pos* et *Neg* d'un *TPredicat*), d'implémenter la fonction *Pt* de l'algorithme générique.

18.3.1 La fonction *CalculerPointFixe*

```
function CalculerPointFixe(Signe :TSigne,P :TPredicat) :TRelation;
begin
    Dg := [ ];
    SommetAppelant := Null;
    for Q in PredTable do
        begin
            Q.Pos := false;
            Q.Neg := false;
        end;
    Calculer(Signe,P);
    if Signe = POS
    then return P.Pos
    else return P.Neg;
end;
```

Cette première fonction **CalculerPointFixe(+,p)**[13] est le point de départ, elle prend une requête $?p$ et renvoie la réponse.

Concrètement, elle initialise le graphe des dépendances et les relations pour chacun des prédicats. Ensuite, elle fait un appel *Calculer(+,p)* pour déclencher le calcul de la relation positive du prédicat p. Une fois cet appel terminé, il ne reste plus qu'à renvoyer le résultat.

[13] Sans nuire à la généralité, nous nous limiterons dans les discussions aux cas où le signe s est positif.

18.3.2 La fonction *Calculer*

```
procedure Calculer(Signe :TSigne,P :TPredicat);
begin
    for S in Dg do
        if (S.Predicat = P) and (S.Signe = Signe)
        then begin
                if (SommetAppelant ≠ NULL) then
                    begin
                        SommetAppelant.Utilise +=S;
                        S.UtilisePar +=SommetAppelant;
                    end;
                return;
            end;
    AncSommetAppelant ←SommetAppelant;
    repeat
        S ←CreateSommet(Signe,P);
        Dg +=S;
        SommetAppelant ←S;
        R ←EvalPredicat(Signe,P);
        if R = false then break;
        if Signe = POS
            then P.Pos ←Ou(P.Pos,R)
            else  P.Neg ←Ou(P.Neg,R);
        if S.UtilisePar = [ ] then break;
        Remove(S);
    end-repeat;
    SommetAppelant ←AncSommetAppelant;
    if (SommetAppelant ≠ NULL) then
        begin
            SommetAppelant.Utilise +=S;
            S.UtilisePar +=SommetAppelant;
        end;
end;
```

La fonction **Calculer(+,p)**, qu'elle soit appelée par la fonction *CalculerPointFixe* ou plus loin par la fonction *EvalLiteral*, a comme objectif de pousser la recherche de la relation positive pour le prédicat p, le plus loin possible.

Une fois ceci terminé, la fonction appelante pourra rechercher le résultat dans la relation $p.Pos$.

Comme dans l'algorithme générique, la première chose que fait la procédure *Calculer(+,p)* est de regarder si le sommet (+,p) se trouve déjà sur le graphe.

Si ceci est le cas, elle sait que soit un calcul de sa valeur est encore en cours, soit que la dernière valeur obtenue ne pourra pas être améliorée par un nouveau calcul. Dans ce cas-ci, la procédure peut donc se terminer après avoir indiqué sur le graphe que le sommet appelant va utiliser la valeur actuelle de $(+, p)$.

Dans le cas contraire, la procédure ajoute le sommet au graphe et fait un appel *EvalPredicat(+,p)* qui renverra l'ensemble des tuples où cette fonction peut dériver que le prédicat est vrai.
On ajoute ces tuples nouvellement trouvés à l'ensemble $P.Pos$.

Si la nouvelle information que l'on vient d'obtenir ainsi, met en cause le résultat de certains sommets, on enlève tous les sommets affectés du graphe et on recommence. Sinon on indique sur le graphe que le résultat obtenu sera utilisé par le sommet appelant et on s'arrête.

18.3.3 La fonction *EvalPredicat*

```
function EvalPredicat(Signe :TSigne,P :TPredicat) :TRelation;
begin
    if Signe = POS
    then begin
            Out ←false;
            for C in P.Conjonctions do
                Out ←Ou(Out,EvalConjonction(POS,C));
         end
    else begin
            Out ←true;
            for C in P.Conjonctions do
                Out ←Et(Out,EvalConjonction(NEG,C));
            end;
    return Out;
end
```

Cette fonction **EvalPredicat**, correspond essentiellement à la fonction $\mathcal{F}_B$ que nous avons vue au paragraphe (7.3) et a comme but d'évaluer la relation positive ou négative d'une disjonction.

18.3.4 La fonction *EvalConjonction*

```
function EvalConjonction(Signe :TSigne,C :TConjonction) :TRelation;
begin
    if (C.FirstTime) then
        begin
            C.ContrainteRelation ←ContraintesToRel(C.Contraintes);
            C.FirstTime ←false;
        end;
    if Signe = POS
    then begin
            Out ←C.ContrainteRelation);
            for L in C.Literaux do
                Out ←Et(Out,EvalLiteral(POS,L));
            Out ←IlExiste(Out,C.VarsQuants);
         end
    else begin
            Out ←Not(C.ContrainteRelation);
            for L in C.Literaux do
                Out ←Ou(Out,EvalLiteral(NEG,L));
            Out ←PourTout(Out, C.VarsQuants);
         end;
    return Out;
end
```

Cette fonction **EvalConjonction**, correspond essentiellement à la fonction $\mathcal{F}_B$ et a comme but d'évaluer la relation positive ou négative d'une *conjonction quantifiée*.

Remarquons que cette fonction veille à ce que la relation correspondante à la *conjonction de contraintes*, ne soit calculée qu'une seule fois puis gardée en mémoire.

La fonction *ContraintesToRel* utilisée ici a été définie au paragraphe (7.2). Son implémentation se trouve à l'intérieur de ce module, mais comme elle ne représente aucun défi particulier, nous n'avons pas repris son pseudo-code.

18.3.5 La fonction *EvalLiteral*

```
function EvalLiteral(Signe :TSigne,L :TLiteral) :TRelation;
begin
    Appel ←L.Appel;
    P ←Appel.Predicat;
    ParEff ←Appel.ParamsEffs;

    if Signe = L.Signe
    then begin
            Calculer(POS,P);
            Out ←P.Pos;
         end
    else  begin
            Calculer(NEG,P);
            Out ←P.Neg;
         end;

    if L.FirstTime then
        begin
            L.FirstTime ←false;
            L.Subst ←[ ];
            for i = 0 to Count(ParEff) do
                if (ParEff[i] is TVar)
                then begin
                        SubstVar.Var ←P.Vars[i];
                        SubstVar.Valeur ←ParEff[i];
                        L.Subst +=SubstVar;
                     end
                else  begin
                        SubstConst.Var ←P.Vars[i];
                        SubstConst.Valeur ←ParEff[i];
                        L.Subst +=SubstConst;
                      end
        end
    Out ←Substituer(Out,L.Subst);
    return Out;
end;
```

La fonction **EvalLiteral(+,L)** correspond essentiellement à la fonction $\mathcal{F}_B$ et a comme but d'évaluer la relation positive ou négative d'un litéral.

Elle est assez coûteuse, puisqu'elle doit effectuer une substitution de variables.
Nous expliquerons à l'aide d'un petit exemple comment nous l'avons implémentée pour ensuite donner quelques suggestions quand à son optimisation.

Considérons donc à titre d'exemple le littéral

$$p(X_1, 3, X_1, X_2),$$

où la définition du prédicat p a la forme suivante :

$$p(X_3, X_4, X_5, X_6) \Leftrightarrow D.$$

Supposons maintenant que l'on ait fait un appel

$$\text{EvalLiteral}(+, p(X_1, 3, X_1, X_2)),$$

qui doit nous renvoyer les tuples de pour lesquels les composants X_1 et X_2 satisfont au littéral.

La première chose que fait la fonction est bien évidemment un appel *Calculer(+,p)*. Le problème qui se pose après est que le résultat $p.Pos$ est exprimé en termes des variables (X_3, X_4, X_5, X_6) de p et doit être traduit en termes des variables (X_1, X_2).

Dans l'exemple considéré, ceci est fait par un appel

$$\text{Substituer}(\text{p.Pos}, [X_3 \to X_1, X_4 \to 3, X_5 \to X_1, X_6 \to X_2]).$$

On rappelle qu'une substitution peut être une opération très efficace ou très coûteuse, selon que la substitution à effectuer est monotone ou pas.

Par exemple, si on a une relation $In(X_1, X_2)$, la substitution de variables

$$[X_1 \rightarrow X_3; X_2 \rightarrow X_4]$$

peut être effectuée en un seul parcours de l'arborescence de la relation *In* en remplaçant pour chaque nœud qu'on rencontre l'étiquette X_1 (resp. X_2) par X_3 (resp. X_4).

Par contre, si pour la même relation $In(X_1, X_2)$ on veut effectuer la substitution

$$[X_1 \rightarrow X_4; X_2 \rightarrow X_3],$$

on ne peut plus se contenter de remplacer les étiquettes des nœuds, puisque en faisant cela, on violerait la contrainte spécifiant que l'étiquette d'un nœud doit être inférieure aux étiquettes de ses fils.
Dans le cas où la substitution des variables n'est pas monotone, une restructuration de l'arborescence s'impose donc.

Il serait intéressant d'examiner quel pourcentage du temps d'exécution de notre interpréteur est perdu en faisant ces restructurations.

Comme une restructuration est une opération très lourde comparée aux autres opérations relationnelles qui peuvent toutes être effectuées en un seul parcours d'arborescence, notre intuition nous dit que ce pourcentage n'est probablement pas négligeable.

Si cette intuition s'avère correcte, il serait intéressant de développer un algorithme de numérotage des variables qui essaie de minimiser le nombre de non-monotonies pour les littéraux.

Par exemple, si on considère le programme suivant :

$$\begin{cases} p(X,Y) := q(Y,X); \\ q(X,Y) := X = Y; \end{cases}$$

notre algoritme de numérotage va numéroter les variables comme suit :

$$\begin{cases} p(X_1,X_2) := q(X_2,X_1); \\ q(X_3,X_4) := X_3 = X_4; \end{cases}$$

avec une non-monotonie dans la substitution $[X_2 \rightarrow X_3; X_1 \rightarrow X_4]$ pour le littéral $q(X_2, X_1)$.
Rien n'empêche pourtant que l'on numérote les variables comme suit

$$\begin{cases} p(X_2,X_1) := q(X_1,X_2); \\ q(X_3,X_4) := X_3 = X_4; \end{cases}$$

ce qui rend la substitution monotone.

Remarquons que, en général, il ne sera pourtant pas possible de lever toutes les non-monotonies (il suffit par exemple de considérer une conjonction telle que $q(X,Y) \wedge q(Y,X)$) et que la conception d'un algorithme de numérotage qui arrive effectivement à minimiser le nombre de non-monoties risque de devenir très complexe.

Rappelons que nous avons implémenté notre logiciel d'une telle manière qu'elle supporte n'importe quel algorithme de numérotage[14]. Ceci facilitera l'intégration de cette optimisation dans de futures versions du logiciel.

[14] Conception modulaire.

18.3.6 Les fonctions *Remove* et *CreateSommet*

```
procedure Remove(S :TSommet) ;
begin
    for S' in S.Utilise
        do S'.UtilisePar −=S ;
    while S.UtilisePar ≠ [ ]
        do Remove(S.UtilisePar[0]) ;
    Dg −=S ;
end ;
function CreateSommet(Signe :TSigne,P :TPredicat,In :TRelation) :TSommet ;
begin
    S.Signe ←−Signe ;
    S.Predicat ←−P ;
    S.In ←−In ;
    S.Utilise ←−[ ] ;
    S.UtilisePar ←−[ ] ;
    return S ;
end ;
```

Les deux dernieres procédures **Remove** et **CreateSommet**, n'ont pas changé par rapport à l'algorithme générique.

18.4 Etendre le graphe des dépendances

Au paragraphe précédent nous avons déjà suggéré une optimisation qui permettrait d'accroître considérablement les performances par rapport à la première version du logiciel que nous avons réalisée. Dans ce chapitre-ci et le suivant, nous aimerions encore faire deux autres suggestions qui vont dans ce sens.

Illustrons la première idée à l'aide du petit programme ci-dessous :

$$\begin{cases} \text{domain}\{a, b\} & \\ p(X_1, X_2) & := \quad q(X_1) \& \; r(X_1, X_2) \;\; | \sim q(X_1) \& \; p(X_2, X_1); \\ q(X_3) & := \quad X_3 = a; \\ r(X_4, X_5) & := \quad X_4 = a \& \; X_5 = a; \end{cases}$$

Traçons en gros ce qui se passe après l'appel

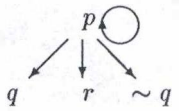
CalculerPointFixe(p).

Après quelques initialisations, cette fonction fait un appel
Calculer(p).

Cette fonction *Calculer(p)*, va exécuter, pour une première fois, l'instruction

R ←EvalPredicat(p).

Après un peu de réflexion, on peut trouver que l'état immédiatement après son exécution sera le suivant : Comme graphe des dépendances, on aura :



Les relations déjà calculées, seront :

$$\begin{cases} \text{Pos}_p = \emptyset & \text{Neg}_p = \emptyset \\ \text{Pos}_q = \{X_3 : a\} & \text{Neg}_q = \{X_3 : b\} \\ \text{Pos}_r = \{X_4 : a, X_5 : a\} & \text{Neg}_r = \emptyset \end{cases}$$

Et finalement, le résultat renvoyé par la fonction *EvalPredicat* sera :

$R = \{X_1 : a, X_2 : a\}$.

La fonction va ensuite exécuter l'instruction **P.Pos ←Ou(P.Pos,R)**, ce qui donne :

$\text{Pos}_p = \{X_1 : a, X_2 : a\}$,

et exécuter l'instruction **Remove(s)** qui va supprimer le sommet p du graphe, de sorte que nous aboutissions au graphe suivant :

Ensuite la fonction doit faire une nouvelle itération, donc rajouter le sommet p au graphe et réexécuter l'instruction **R ←EvalPredicat(p)**.

Ce que nous voulons ilustrer ici est que dans cette deuxième exécution, on va réévaluer beaucoup trop. En effet, si on mesure la complexité de l'algorithme en termes du nombre d'appels à la fonction *EvalLiteral* (assez coûteux à cause de la substitution de variables) et le nombre d'opérations relationnelles, on observe qu'on ferra

- quatres appels à *EvalLiteral* :

  EvalLiteral($q(X_1)$)
  EvalLiteral($r(X_1, X_2)$)
  EvalLiteral($\sim q(X_1)$)
  EvalLiteral($p(X_2, X_1)$)

- deux opérations relationelles *Et* dans respectivement

  EvalConjonction($q(X_1)$ & $r(X_1, X_2)$)
  EvalConjonction($\sim q(X_1)$ & $p(X_2, X_1)$)

- une opération relationelle *Ou*, dans EvalPredicat(p).

Comme tout ce qui a changé par rapport à la dernière exécution est la relation *p.Pos*, on voit que les trois premiers appel de *EvalLiteral* ainsi que la première opération *Et*, sont parfaitement inutiles.

On a mis le doigt sur une inefficacité de notre version de l'algorithme qui sera d'autant plus grave, que les formules du programme soient plus longues.

Nous voulons dans la suite de ce paragraphe, proposer une adoptation de l'algorithme qui permettrait d'éviter toutes les opérations inutiles que nous venons de citer.

L'idée de base est qu'on va mettre dans le graphe des dépendances, non seulement les prédicats, mais aussi les conjonctions et les litéraux.

Pour gérer ce graphe étendu, on aura besoin de deux nouvelles fonctions :

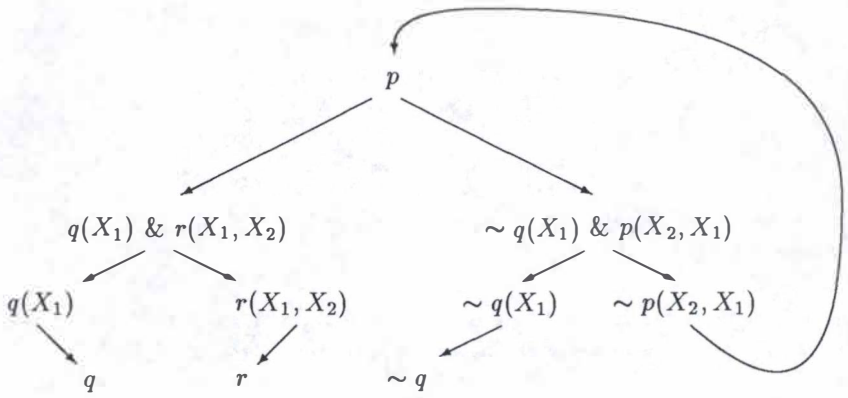
function CalculerConjonction(S :TSigne,C :TConjonction)
: TRelation

function CalculerLiteral(S :TSigne,L :TLiteral) : TRelation

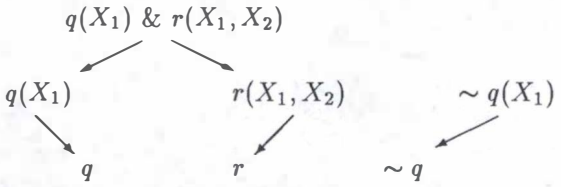
On rappelle que la fonction *Calculer(p)* a comme objectif de vérifier si le prédicat p se trouve déjà sur le graphe, et ne va appeler *EvalPredicat(p)* que si ceci n'est pas le cas.
Les fonctions *CalculerConjonction(C)* (resp. *CalculerLiteral(L)*) vont faire exactement la même chose, mais alors pour les conjonctions (resp. literaux) au lieu des prédicats.
La fonction *EvalPredicat* ne va plus directement appeler *EvalConjonction*, mais va faire un appel à *CalculerConjonction* de sorte qu'elle puisse, après cet appel, récupérer le résultat dans un attribut *C.Pos*

de la conjonction C en question.

Selon cette nouvelle aproche, on a, après la première exécution de **R ←EvalPredicat(p)**, le graphe de dépendences suivant :



R étant différent de $\emptyset$, on ajoute alors R à la relation $p.Pos$, pour ensuite, sur le graphe des dépendences, supprimer tout ce qui dépend de l'ancienne valeur.
On aboutit alors au graphe suivant :



Auquel on rajoute le sommet p avant de faire une deuxième exécution de
R ←EvalPredicat(p).
Une petite réflexion démontre que cette manière on évite toutes les opérations inutiles citées ci-dessus.

18.5 Une version orientée requête

Une dernière suggestion quant à l'optimisation de notre interpréteur peut être illustrée à l'aide de l'exemple suivant :

$$\begin{cases} p(X_1) & := & q(a, X_1); \\ q(X_2, X_3) & := & X_2 <> a \ \& \ r(X_2, X_3); \\ r(X_4, X_5) & := & \cdots \\ \cdots \end{cases}$$

Si l'utilisateur fait une requête $?p$, l'interpréteur doit calculer la partie positive de p, pour ceci il doit lancer le calcul de la partie positive de q, ce qui entraîne à son tour le calcul de partie positive de r.
Pourtant, si on regarde bien le système d'équations, on remarque que l'on n'a pas besoin de connaître tous les tuples (X_2, X_3) pour lesquels q est vrai, mais que l'on peut d'avance se limiter aux tuples (X_2, X_3) pour lesquels $X_2 = a$.
Si on savait incorporer ceci dans l'algorithme, on pourrait déjà après la contrainte $X_2 <> a$, décider qu'il n'existe pas de tels tuples, sans que l'on doive pour cela calculer quoi que ce soit pour le prédicat r.

On donne encore un deuxième exemple, un peu plus subtil :

$$\begin{cases} p(X_1, X_2) & := & q(X_1, X_2) \ \& \ r(X_1, X_2); \\ q(X_3, X_4) & := & X_3 = X_4; \\ r(X_5, X_6) & := & X_5 <> X_6 \ \& \ s(X_5, X_6); \\ s(X_7, X_8) & := & \cdots \\ \cdots & & \end{cases}$$

A nouveau, suite à la requête $?p$, l'algorithme va lancer le calcul de la partie positive de s.
Pourtant, une fois que l'algorithme a trouvé les tuples (X_1, X_2) pour lesquels l'appel $q(X_1, X_2)$ est vrai, il ne devrait plus essayer de trouver tous les tuples qui satisfont $r(X_1, X_2)$, mais il pourrait se limiter aux tuples déjà trouvés.
Comme parmi ces tuples, il n'en existe aucun pour lequel $X_1 <> X_2$, il pourrait déjà après la contrainte $X_5 <> X_6$, décider qu'il n'existe pas de solution et éviter ainsi le calcul du point fixe en s.

Les problèmes évoqués ici, découlent principalement du fait que la fonction $\mathcal{F}_B$ tel que nous l'avons définie, n'est pas assez fine, dans le sens où elle ne permet pas de restreindre l'évaluation d'une sous-formule à un sous-ensemble du domaine.

La solution à ce problème consiste en l'élaboration, encore, d'une nouvelle formulation de la sémantique.

Dans cette nouvelle version on va considérer une interprétation d'un programme P comme une fonction

$$\varphi : \{+, -\} \times \{p_0, \ldots, p_\nu\} \times \mathcal{SR} \longrightarrow \mathcal{SR}$$

qui associe à un signe s, un prédicat p et une relation d'entrée In, une relation de sortie, qui contient tous les tuples de In pour lesquels on sait que le prédicat p est vrai (si $s = +$) ou faux (si $s = -$).

Notons l'ensemble de ces interprétations par ω_P.

On peut alors définir un opérateur $\mathcal{G}$ qui décrit la sémantique d'une sous-formule.

Cet opérateur va, étant donné une interprétation $\varphi \in \omega_P$, un signe s, une sous-formule f et une relation d'entrée In, renvoyer une relation $\mathcal{G}^s_\varphi(f, \mathrm{In})$.

Intuitivement, $\mathcal{G}^+_\varphi(f, \mathrm{In})$ (resp. $\mathcal{G}^-_\varphi(f, \mathrm{In})$) est l'ensemble des tuples dans In, où on sait déduire (en se basant sur les connaissances partieles des prédicats dans φ) que la sous-formule f est vraie (resp. fausse).

De façon plus formelle, on peut définir cet opérateur comme suit :

- Pour un appel de prédicat :

  Si la définition du prédicat a la forme $p(X_{i_1}, \ldots, X_{i_n}) := D$, alors :

  $$\mathcal{G}^s_\varphi(p(e_1, \ldots, e_n), \mathrm{In}) = \\ \quad \mathrm{Substituer}(\varphi(s, p, \mathrm{In'}), [X_{i_1} \to e_{i_1}, \ldots, X_{i_n} \to e_{i_n}]),$$

  où In' est la relation In (qui est exprimée en termes des variables de $e_1, \ldots, e_n$), transformée en une relation en termes des variables $X_{i_1}, \ldots, X_{i_n}$.
- Pour un litéral :

  Soit A un appel de prédicat, alors :

  $$\mathcal{G}^+_\varphi(\neg A, \mathrm{In}) = \mathcal{G}^-_\varphi(A, \mathrm{In}) \\ \mathcal{G}^-_\varphi(\neg A, \mathrm{In}) = \mathcal{G}^+_\varphi(A, \mathrm{In})$$

- Pour une conjonction de literaux :

Soit $L_1, \ldots, L_n$ des litéraux, alors :

$$\mathcal{G}^+_\varphi(L_1 \wedge L_2 \wedge \ldots \wedge L_n, \mathrm{In}) = \mathcal{G}^+_\varphi(L_2 \wedge \ldots \wedge L_n, \mathcal{G}^+_\varphi(L_1, \mathrm{In}))$$

$$\mathcal{G}^-_\varphi(L_1 \wedge \ldots \wedge L_n, \mathrm{In}) = \mathcal{G}^-_\varphi(L_1, \mathrm{In}) \vee \ldots \vee \mathcal{G}^-_\varphi(L_n, \mathrm{In})$$

- Pour une conjonction :

  Soit C une conjonction de contraintes, L une conjonction de literaux :
  - $\mathcal{G}^s_\varphi(L, \mathrm{In})$ est déjà défini.
  - $\mathcal{G}^+_\varphi(C, \mathrm{In}) = \mathrm{ContraintesToRel}(C) \wedge \mathrm{In}$
    $\mathcal{G}^-_\varphi(C, \mathrm{In}) = \neg\ \mathrm{ContraintesToRel}(C) \wedge \mathrm{In}$
  - $\mathcal{G}^+_\varphi(C \wedge L, \mathrm{In}) = \mathcal{G}^+_\varphi(L, R \wedge \mathrm{In})$,
    $\mathcal{G}^-_\varphi(C \wedge L, \mathrm{In}) = (\mathrm{Not}(R) \wedge \mathrm{In}) \vee \mathcal{G}^-_\varphi(L, \mathrm{In})$,
    où $R = \mathrm{ContraintesToRel}(C)$.

- Pour une conjonction quantifiée :

  Soit C une conjonction, alors :

$$\mathcal{G}^+_\varphi(\exists X_{i_1}, \ldots, X_{i_n} : C, \mathrm{In}) = \mathrm{IlExiste}(\mathcal{G}^+_\varphi(C, \mathrm{In}), [X_{i_1}, \ldots, X_{i_n}]),$$

$$\mathcal{G}^-_\varphi(\exists X_{i_1}, \ldots, X_{i_n} : C, \mathrm{In}) = \mathrm{PourTout}(\mathcal{G}^-_\varphi(C, \mathrm{In}), [X_{i_1}, \ldots, X_{i_n}]),$$

- Pour une disjonction :

  Soit $C_1, \ldots, C_n$ des conjonctions quantifiées, alors :

$$\mathcal{G}^-_\varphi(L_1 \vee L_2 \vee \ldots \vee L_n) = \mathcal{G}^-_\varphi(L_2 \vee \ldots \vee L_n, \mathcal{G}^-_\varphi(L_1, \mathrm{In}))$$

$$\mathcal{G}^+_\varphi(L_1 \vee \ldots \vee L_n) = \mathcal{G}^+_\varphi(L_1, \mathrm{In}) \vee \ldots \mathcal{G}^+_\varphi(L_n, \mathrm{In})$$

Muni de cet opérateur $\mathcal{G}$, on peut alors définir l'opérateur XS_P qui prend une interprétation $\varphi \in \omega_P$ comme approximation de la sémantique et qui renvoie l'approximation suivante.

Cet opérateur

$$\mathrm{XS}_P : \omega_P \to \omega_P$$

est défini comme suit :

Supposons que la définion du prédicat p ait la forme

$$p(X_{i_1}, \ldots, X_{i_n}) \coloneqq D,$$

alors

$$(\mathrm{XS}_P\ \varphi)(s, p, \mathrm{In}) = \mathcal{G}^s_\varphi(p, \mathrm{In}).$$

La sémantique du programme P peut alors être définie comme le plus petit point fixe de cet opérateur et la réponse à une requête ? p (resp. ? $\neg\ p$) comme

$$(\mu\ \mathrm{XS}_P)(+, p, U) \text{ (resp. } (\mu\ \mathrm{XS}_P)(-, p, U) \text{),}$$

où U est le domaine de p.

Avec cette nouvelle formulation de la sémantique, on pourrait alors appliquer l'algorithme de calcul de point fixe en prenant comme

- ensemble A, l'ensemble

  $$\{+, -\} \times \{p_0, \ldots, p_\nu\} \times \mathcal{SR},$$

  des signes, prédicats et relations d'entrée
- treillis complet A', l'ensemble $\mathcal{SR}$ des relations

- opérateur τ, l'opérateur XS_P,
- fonction effective, la fonction définie par :

  $$\mathrm{tau}(\varphi, (s, p, \mathrm{In})) = \mathcal{G}^s_\varphi(D, \mathrm{In}),$$

  où D est la disjonction dans la définition de p.

Cette nouvelle formulation du problème permettrait à notre interpréteur de n'évaluer que les parties des prédicats dont il a effectivement besoin pour fournir la réponse.

Nous n'avons plus eu le temps d'implémenter ces idées, mais à notre avis les gains au niveau du temps de réponse et au niveau de la mémoire qui peuvent en résulter, pourrait s'avérer considérable pour certaines applications.

Remarquons encore que plusieurs optimisations s'imposent si on veut implémenter le schéma que nous venons d'esquisser.

- On pourrait par exemple définir une relation d'ordre sur l'ensemble

  $$A = \{+, -\} \times \{p_0, \ldots, p_\nu\} \times \mathcal{SR},$$

  comme suit :

  $$(s, p, \mathrm{In}) \sqsubseteq (s', p', \mathrm{In}') \Leftrightarrow s = s' \wedge p = p' \wedge \mathrm{In} \subseteq \mathrm{In}'.$$

  Nous référons à l'article [1] pour une explication sur comment une telle relation d'ordre peut être exploitée pour optimiser le calcul du point fixe.
- Une autre approche (incompatible avec celle que nous venons de citer) serait de considérer que si un sommet $(+, p, In)$ se trouve déjà sur le graphe des dépendances et une demande d'évaluation de la partie positive de p dans un ensemble In est faite, il n'est plus nécessaire de réévaluer p dans l'ensemble $In \cap In_0$.
  On pourrait en effet restreindre la nouvelle évaluation à l'ensemble $In \backslash In_0$, si on indique sur le graphe des dépendances que le sommet appelant va utiliser, non seulement le nouveau sommet

  $$(+, p, \mathrm{In} \backslash In_0),$$

  mais également l'ancien sommet

  $$(+, p, In_0).$$

Des tests devraient démontrer laquelle des deux approches est la mieux adaptée à notre problème.

19 L'affichage d'une relation

module printrelation.c
interface : **PrintRelation** : TRelation →
utilise : tables.c trelation.c tliste.c

Ce module exporte une seule fonction **PrintRelation** qui permet d'afficher une relation sous un format compréhensible par l'utilisateur.

La représentation interne de la relation (qui est, on le rapelle, faite sous forme d'un arbre), est interprétée comme une table et les numéros de variables ou de constantes sont reconvertis en forme textuelle.

20 Le module coordinateur

module principal.c
interface : **main** : String × String → main(NomFichier,Requete)
utilise : parse.y numerotage.c typage.c pointfixe.c printrelation.c

Ce module "coordinateur" contient une seule procédure **main** qui prend un nom de fichier et une requête en argument et qui va faire tous les appels nécessaires à la résolution de cette requête.

Pour complétude, nous ajoutons le pseudo-code de cette procédure :

```
procedure main(NomFichier, Requete : String)
begin
    InitTables();
    InitRelationTable(20000);
    yyin ←-open(NomFichier,lecture);
    yyparse();
    NumeroterPredicats();
    NumeroterVars();
    Typage();
    if (Requete[0] = '+')
    then R ←CalculerPointFixe(POS,Requete+1)
    else
        if (Requete[0] = '-')
        then R ←CalculerPointFixe(NEG,Requete+1)
        else  R ←CalculerPointFixe(POS,Requete);
    PrintNoeud(R);
end.
```

21 Les listes

module principal.c
interface :
utilise :

TListe : SEQ[T]

Les listes génériques

- **CreateListe** : → SEQ[T]
  L=CreateListe()
- **DestroyListe** : SEQ[T] →
  DestroyListe(L)
  post: détruit la liste, ainsi que tous les éléments stockés dedans
- **Count** : SEQ[T] → Integer
  i = Count(L)
- **GetItem** : SEQ[T] × Integer → T
  x = GetItem(L,i)
  pré : $0 \leq i < Count(L)$
  post: x est l'élément L[i] (et *pas* une copie de celui-ci)
- **GetLast** : SEQ[T] → T
  x = GetLast(L)
  pré : $0 < Count(L)$
  post: x est le dernier élément de L(et *pas* une copie de celui-ci)
- **Extract** : SEQ[T] × Integer → T
  x = Extract(var L, i)
  pré : $0 \leq i < Count(L)$
  post: x est l'élément L[i] (et *pas* une copie de celui-ci)
  cet élément est extrait de la liste
- **ExtractLast** : SEQ[T] → T
  x = ExtractLast(var L)
  pré : $0 < Count(L)$
  post: x est l'élément L[i] (et *pas* une copie de celui-ci)
  cet élément est extrait de la liste
- **GetIndex** : SEQ[T] × T → Integer
  i = GetIndex(L,x)
  post: i est l'index de la première occurence de x dans L
  et -1 si x n'apparaît pas dans L
  remarque : le test d'égalité porte seulement sur l'adresse de x, pas sur son contenu
- **Add** : SEQ[T] × T →
  Add(var L, x)
  post: x (et *pas* une copie) est ajouté à la fin de la liste
- **AddToSet** : SEQ[T] × T →
  AddToSet(var L, x)
  post: x est seulement ajouté à L s'il n'y apparaît pas encore
- **Insert** : SEQ[T] × T × Integer →
  Insert(var L, x, i)
  pré : $0 \leq i \leq Count(L)$
  post: x est inséré à la position i

- **Swap** : SEQ[T] × Integer × Integer →
  Swap(var L, i, j)
  pré : 0 ≤ i,j < Count(L)
  post: les éléments L[i] et L[j] sont échangés
- **Reset** : SEQ[T] →
  Reset(var L)
  post: le premier élément de la liste devient l'élément courant
- **EOL** : SEQ[T] → Boolean
  b = EOL(L)
  post: b est true si l'élément courant de L pointe après son dernier élément
- **GetNext** : SEQ[T] → T
  x = GetNext(var L)
  pré :!EOL(L)
  post: x est l'élement courant de L et l'élément courant de L est avancé de 1

Les listes d'entiers

- **GetInt** : SEQ[Integer] → Integer → Integer
  x = GetInt(L,i)
  pré : 0 ≤ i < Count(L)
- **ExtractInt** : SEQ[Integer] × Integer → Integer
  x = ExtractInt(var L, i)
  pré : 0 ≤ i < Count(L)
- **GetIndexInt** : SEQ[Integer] × Integer → Integer
  i = GetIndexInt(L,x)
  post: i est l'index de la première occurence de x dans L
  et -1 si x n'apparaît pas dans L
- **AddToIntListe** : SEQ[Integer] × Integer →
  AddToIntSet(var L, x)
- **AddToSortedIntSet** : SEQ[Integer] × Integer →
  AddToSortedIntSet(var L, x)
  pré : L est trié et ne contient pas de doubles
  post: si x n'apparaît pas dans L, il y est ajouté à la position correcte
- **AppendIntListe** : SEQ[Integer] × SEQ[Integer] → SEQ[Integer]
  C = AppendIntListe(A,B)
- **CopyIntListe** : SEQ[Integer] → SEQ[Integer]
  B = CopyIntListe(A)
- **SortIntListe** : SEQ[Integer] →
  SortIntListe(var L)

Les listes de chaînes de charactères

- **GetString** : SEQ[String] × Integer → Integer
  s = GetString(L,i)
  pré : 0 ≤ i < Count(L)
  s est la i-ième chaîne dans L (et *pas* une copie)
- **GetIndexString** : SEQ[String] × String → Integer
  i = GetIndexString(L,s)
  post: i est l'index de la première occurence de s dans L
  et -1 si s n'apparaît pas dans L
  remarque : il s'agît ici d'une égalité de chaînes de caractères, et pas d'une égalité d'adresse

- **AddCopyToStringListe** : SEQ[String] × String →
  AddCopyToStringListe(var L,s)
  post: une *copie* de s est ajoutée à la fin de L
- **AddCopyToStringSet** : SEQ[String] × String →
  AddCopyToStringListe(var L,s)
  post: si $s \notin L$, une *copie* de s est ajoutée à la fin de L
- **AddToStringSet** : SEQ[String] × String →
  AddToStringSet(var L, s)
  post: s même (et *pas* une copie) est ajouté à la fin de L
- **AppendStringListe** : SEQ[String] × SEQ[String] → SEQ[String]
  C = AppendStringListe(A,B)
  post: C est le résultat de la concaténation de A et de B
  les chaînes dans C sont des *copies* des chaînes dans A et B
- **DiffStringList** : SEQ[String] × SEQ[String] → SEQ[String]
  C = DiffStringList(A,B)
  post: $C = \{s \mid s \in A \wedge s \notin B\}$
  les chaînes dans C sont des *copies* des chaînes dans A et B

Troisième partie
Le logiciel

22 Invocation

L'interpréteur de notre langage possède une invocabilité résumée à sa plus simple expression. Si aucun paramètre ne lui est transmis, il renvoie à l'utilisateur les précisions suivantes sur l'utilisation du programme :

usage : beta −f nom_fichier −p ['+'|'−'|'']nom_predicat [−t table_size]

Ainsi

- le paramètre **obligatoire** −f doit être suivi du nom du fichier source que l'interpréteur devra lire ;
- le paramètre **obligatoire** −p doit être suivi du nom du prédicat dont on recherche la solution. Si ce prédicat est absent du fichier source, un message d'erreur est renvoyé. Dans le cas où le nom de prédicat est précédé du signe +, l'utilisateur recevra comme réponse l'ensemble des tuples pour lesquels le prédicat est vrai. Dans le cas où le nom de prédicat est précédé du signe −, l'utilisateur recevra comme réponse l'ensemble des tuples pours lesquels le prédicat est faux. Si le signe est omis, on retombe par défaut dans le cas du signe + ;
- le paramètre **optionnel** −t doit être suivi d'un entier. Il spécifie explicitement la taille du tableau de hashing à réserver en mémoire avant l'interprétation du code source. Ce nombre peut être aussi grand que désiré, tout en restant dans les limites de l'allocabilité du système. Si ce prarmètre est omis, la taille du tableau est par défaut étalonnée pour vingt mille nœuds.

Exemples :

```
beta −f mon_fichier.x −p nabuchodonosor

beta −f tralala.x −p johnny_guitar −t 100000
```

Toutefois, les invocations suivantes seront refusées :

```
beta −f mon_fichier.x −p nabuchodonosor(X,Y)        (variables dans le prédicat)

beta −f mon_fichier.x                               (prédicat à résoudre non spécifié)
```

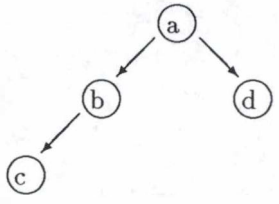
23 Exemples d'exécution

En plus de l'exécution des programmes, nous allons critiquer quelques peu les paramètres qui interviennent dans cette affaire.
Nous avons implémenter avec notre langages trois exemples précis :

- l'arbre généalogique cité en exemple plus tôt ;
- le problème du loup de la chèvre et du chou ;
- le problème des huit reines, et plus génériquement des n reines.

23.1 Description des problèmes

L'arbre généalogique De conception très simple, il se résume en un arbre :



Pour cela, nous avions rédigé le programme suivant :

```
Pere(X,Y)     :=  X=a & Y=b
                | X=b & Y=c
                | X=a & Y=d;
Ancetre(X,Y)  :=  Pere(X,Y)
                | Pere(X,Z) & Ancetre(Z,Y);
```

La loup, la chèvre et le chou [15]
Très classique et vu très énigmatiquement par les plus jeunes des bancs de l'école primaire, son énoncé est aussi, somme toute, très simple :

Comment le batelier parviendra-t-il à déposer sur l'autre rive, sains et saufs, le loup, la chèvre et le chou, sachant qu'il ne peut transporter qu'une personne ou chose dans son embarcation, que le loup ne peut rester seul avec la chèvre et que la chèvre ne peut rester seule avec le chou[16] *?*

Ceci fait l'objet du code suivant :

```
domain {left,right};

danger(M,W,G,C) := W=G & M<>W | G=C & M<>G;

transition(M0,W0,G0,C0,M,W,G,C) :=
                | M<>M0 & W=W0 & G=G0 & C=C0
                | M<>M0 & W<>W0 & G=G0 & C=C0 & M=W
                | M<>M0 & W=W0 & G<>G0 & C=C0 & M=G
                | M<>M0 & W=W0 & G=G0 & C<>CO & M=C;

reachable(M,W,G,C) :=
                  M=left & W=left & G=left & C=left
                | not danger(M,W,G,C) & transition(M0,W0,G0,C0,M,W,G,C);
                                      & reachable(M0,W0,G0,C0);

p() := reachable(right,right,right,right);
```

[15] En anglais : *The Wolf, The Goat and The Cabbage.*
[16] Sinon, le loup mange la chèvre, et la chèvre mange le chou.

Le domaine est binaire : soit on est sur la rive gauche, soit on est sur la rive droite. Les variables de ce programme se réfèrent toutes à ce domaine par défaut car l'important est de savoir où le batelier (M), le loup (W), la chèvre (G) et le chou (C) se trouvent à un instant donné. Quant aux prédicats, les trois premier formalisent les éléments qui contraignent le problème :

- quelles sont les situations de danger ?
- peut-on passer de n'importe quelle situation à n'importe quelle situation ?
- une configuration donnée est-elle possible par la suite sachant que tous les quatre attendent sur la rive gauche (situation de départ) ?

Les situations de danger sont implicitement formulées dans l'énoncé :

- le loup ne peut rester seul avec la chèvre, donc on ne peut laisser le loup et la chèvre sur la même rive et laisser l'homme sur une rive différente de celle du loup ;
- la chèvre ne peut rester seule avec le chou, donc on ne peut laisser la chèvre et le chou sur la même rive et laisser l'homme sur une rive différente de celle de la chèvre.

Ce qui explique le prédicat danger :

```
danger(M,W,G,C) := W=G & M<>W | G=C & M<>G ;
```

En dehors de cela, il faut aussi formuler les transformations possibles, c'est-à-dire comment passer d'une situation à une autre. Nos quatre variables M, W, G, C peuvent en effet changer selon ce que la batelier emmènera de l'autre côté de la rive. Quatre mouvements sont possibles :

- le batelier part seul de l'autre côté ;
- le batelier emmène le loup avec lui si ce dernier est sur le même rive ;
- le batelier emmène la chèvre avec lui si cette dernière est sur le même rive ;
- le batelier emmène le chou avec lui si ce dernier est sur le même rive.

Ce qui justifie le prédicat suivant :

```
transition(M0,W0,G0,C0,M,W,G,C) :=
                | M<>M0 & W=W0 & G=G0 & C=C0
                | M<>M0 & W<>W0 & G=G0 & C=C0 & M=W
                | M<>M0 & W=W0 & G<>G0 & C=C0 & M=G
                | M<>M0 & W=W0 & G=G0 & C<>CO & M=C ;
```

où M0, W0, G0, C0 désignent un état inial et M, W, G, C désignent un état final.
Les deux prédicats danger et transition peuvent être combinés en un troisième pour savoir si un état est légitime ou illégitime. Nous avons appelé ce prédicat reachable(M,W,G,C), car il jauge un état et dit s'il est atteignable. Il existe deux cas de figure :

- tous les acteurs attendent sur la rive gauche ;
- l'état ne présente aucun danger et provient d'un état accessible.

Ainsi, nous écrivons :

```
reachable(M,W,G,C) :=
            M=left & W=left & G=left & C=left
          | not danger(M,W,G,C) & transition(M0,W0,G0,C0,M,W,G,C)
                                & reachable(M0,W0,G0,C0) ;
```

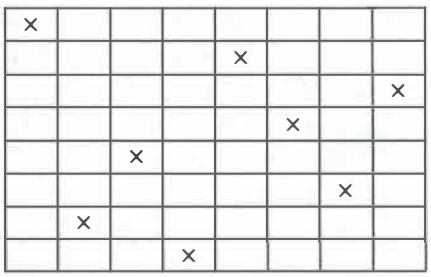
L'option choisie pour obtenir la solution au problème, est de poser un prédicat bidon :

```
p() := reachable(right,right,right,right) ;
```

Le problème des n reines L'énoncé est le suivant :

Sachant qu'une reine sur un échiquier prend toute pièce sur la même horizontale, sur la même verticale, et sur les mêmes diagonales qu'elle, comment placer huit reines sur un échiquier sans qu'elles ne se prennent ?

Voici un exemple de solution, parmi les 92 configurations satisfaisantes :



De façon générale : *comment est-il possible de placer n reines sur un échiquier à n lignes et n colonnes ?* La formulation en β-langage repose sur quelques étapes que mêmes les plus médiocres amateurs d'échecs auront vite fait de parcourir :

- nous avons déclaré un domaine par défaut qui est un intervalle d'entiers {1,2,3,4,5,6,7,8}. Le but est de considérer huit variables définies sur ce domaine. Ainsi, chaque colonne de l'échiquier possède sa variable, et si par exemple, la variable de la colonne 5 reçoit la valeur 2, cela signifie que la reine de cette colonne sera placée sur la seconde ligne. La grille citée en exemple sera codée par $C_1 = 8,\ C_2 = 2,\ C_3 = 4,\ C_4 = 1,\ C_5 = 7,\ C_6 = 5,\ C_7 = 3,\ C_8 = 6$;

```
domain a [1..8];
```

- afin de poser un peu d'ordre sur cet intervalle, on définit un prédicat succ(X,Y) qui explique quels sont les entiers qui se suivent. Cela est nécessaire car notre langage ne manipule pas d'ordre implicite sur les éléments d'un domaine :

```
succ(X,Y)  :=  X=a[1] & Y=a[2]  |  X=a[2]&Y=a[3]  |  X=a[3]&Y=a[4]
           |   X=a[4] &Y=a[5]   |  X=a[5]&Y=a[6]  |  X=a[6]&Y=a[7]
           |   X=a[7] &Y=a[8]   ;
```

- sur cette base, nous pouvons formaliser le fait qu'une reine soit sur la même diagonale qu'une autre. Toutefois, nous réintroduisons ici ligne et colonne : la position (L_1, C_1) est sur la même diagonale que la position (L_2, C_2) si L_2 est successuer de L_1 et C_2 est successeur de C_1 ou alors si il existe un tiers élément de position (L_3, C_3) tel que (L_3, C_3) soit sur la diagonale de (L_1, C_1) et dont (L_2, C_2) en est la successeur. Attention, ce prédicat, de par sa formulation, permet seulement de dire si (L_2, C_2) est sur la même diagonale que (L_1, C_1), mais pas l'inverse !

```
diag_db(L1,C1,L2,C2)  :=  succ(L1,L2) & succ(C1,C2)
                      |   diag_db(L1,C1,L3,C3) & succ(L3,L2) & succ(C3,C2);
```

- parler d'une prise entre deux reines devient à présent facile : elle ne peuvent être sur la même ligne, ni sur la même colonne, ni sur la même diagonale. Comme le prédicat diag_db(L1,C2,L2,C2) ne marche que dans un sens, il ne faut pas oublier de le réappeler en permutant des arguments :

```
prise(L1,C1,L2,C2)  :=  L1=L2 | C1=C2
                    |   diag_db(L1,C1,L2,C2) | diag_db(L2,C2,L1,C1)
                    |   diag_db(C1,L2,C2,L1) | diag_db(C2,L1,C1,L2);
```

  Vus différemment, les quatre appels correspondent au fait que d'une reine peuvent partir quatre bouts de diagonales.

- enfin, nous pouvons formaliser ce qu'est une solution par un raisonnement en cascade, comme lors d'un placement de pièces sur un échiquier où l'on place la première, puis la seconde sans empiéter sur la première, puis la troisème, etc. On remarquera que ce prédicat n'admet pour paramètres que les variables associées aux colonnes. Les appels aux autres prédicats recoivent des numéros de lignes déductibles par la structure du domaine. De plus, tous les appels de ce prédicat sont précédés d'un not.

```
sol(C1,C2,C3,C4,C5,C6,C7,C8)  :=  not prise(a[1],C1,a[2],C2) &

    not prise(a[1],C1,a[3],C3)  &  not prise(a[2],C2,a[3],C3) &

    not prise(a[1],C1,a[4],C4)  &  not prise(a[2],C2,a[4],C4) &
    not prise(a[3],C3,a[4],C4)  &

    not prise(a[1],C1,a[5],C5)  &  not prise(a[2],C2,a[5],C5) &
    not prise(a[3],C3,a[5],C5)  &  not prise(a[4],C4,a[5],C5) &

    not prise(a[1],C1,a[6],C6)  &  not prise(a[2],C2,a[6],C6) &
    not prise(a[3],C3,a[6],C6)  &  not prise(a[4],C4,a[6],C6) &
    not prise(a[5],C5,a[6],C6)  &

    not prise(a[1],C1,a[7],C7)  &  not prise(a[2],C2,a[7],C7) &
    not prise(a[3],C3,a[7],C7)  &  not prise(a[4],C4,a[7],C7) &
    not prise(a[5],C5,a[7],C7)  &  not prise(a[6],C6,a[7],C7) &

    not prise(a[1],C1,a[8],C8)  &  not prise(a[2],C2,a[8],C8) &
    not prise(a[3],C3,a[8],C8)  &  not prise(a[4],C4,a[8],C8) &
    not prise(a[5],C5,a[8],C8)  &  not prise(a[6],C6,a[8],C8) &
    not prise(a[7],C7,a[8],C8)  ;
```

23.2 Mesures de l'exécution

Plusieurs paramètres ont été choisis. Voici leurs définitions :

- **Temps d'exécution** ;
- **MAX_NOEUDS** : le nombre de nœuds supportés par la table de hashing ;
- **Nœud_Table_Count** : le nombre de nœuds effectifs dans la table. Cette valeur est mesurée après l'exécution de l'algorithme, quand tout est fini ;
- **Max_Nœud_Table_Count** : le nombre maximal de nœuds utilisés à un moment de l'exécution ;
- **Facteur de partage** : il s'agit d'un rapport entre les nœuds soumis à un partage (cfr section 10.3 page 69) et les nœuds sans partage ;
- **Mal placés** : recensement des nœuds qui n'ont pu être placés à l'adresse que la fonction de hashing leur avait assignée ;
- **Distance moyenne** : distance moyenne séparant un nœud de son adresse de base (adresse calculée par la fonction de hashing) ;
- **Distance maximale** : le plus grand des éloignements survenu entre un nœuds et son adresse de base.

Ces tests ont été menés sur un processeur *InHell Pençium* 200 muni d'une technologie *MMX*. Il était équipé de 32 mégabytes de mémoire vive. Le système d'exploitation était **Linux**. Le temps d'occupation du processeur par un processus peut dans ce cadre être mesuré par le biais de l'instruction **time** qui reçoit comme argument le nom du programme à exécuter ainsi que tous les paramètres nécessaires à ce programme.

23.3 Résultats

L'arbre généalogique L'exécution de ce programme fut très brève : 0.01 seconde ! La solution qu'elle retourne est celle du prédicat ancetre(X,Y) :

```
(X : a, Y : b)
(X : a, Y : c)
(X : a, Y : d)
(X : b, Y : c)
```

Les paramètres renvoyés sont

Temps	0.01
MAX_NOEUDS	20000
Nœuds_Table_Count	9
Max_Nœuds_Table_Count	18
Facteur de partage	2.44
Mal placé	0
Distance moyenne	0.00
Distance maximale	0

Le loup, la chèvre et le chou Le temps de résolution est aussi extrêment court : 0.02 secondes. La solution renvoyée est celle du prédicat p(). Sa réponse est :

TRUE

Les paramètres renvoyés sont

Temps	0.02
MAX_NOEUDS	20000
Nœuds_Table_Count	134
Max_Nœuds_Table_Count	223
Facteur de partage	4.94
Mal placé	0
Distance moyenne	0.00
Distance maximale	0

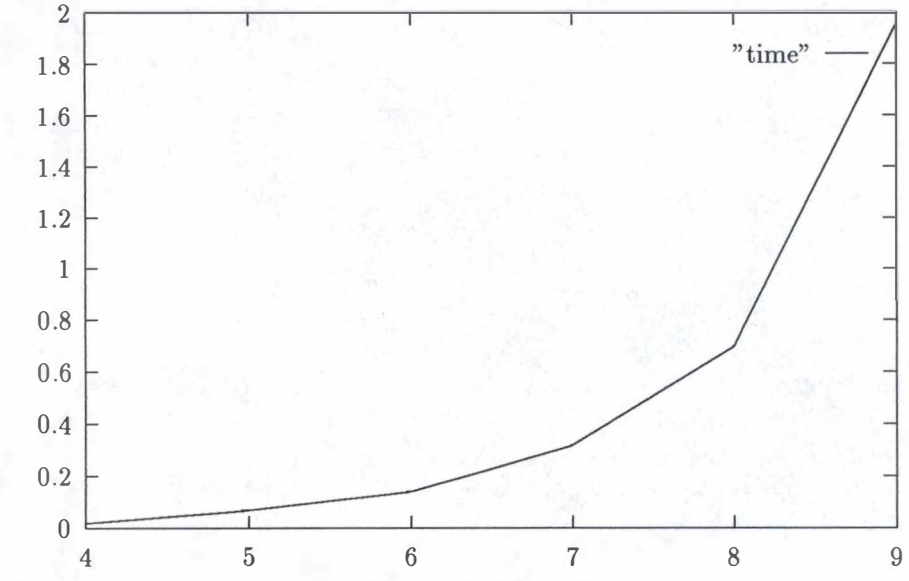
Le problème des n reines Evidemment, pour $n \in \{0, 1, 2, 3\}$, ce problème n'admet aucune solution.

	$n = 4$	$n = 5$	$n = 6$	$n = 7$	$n = 8$	$n = 9$	$n = 10$[17]
Temps	0.02	0.07	0.14	0.32	0.70	1.95	
Nombre de solutions	2	10	4	40	92	352	724
MAX_NOEUDS	20000	20000	20000	20000	20000	20000	
Nœuds_Table_Count	152	299	495	964	2070	6326	
Max_Nœuds_Table_Count	162	325	610	1262	3024	9480	
Facteur de partage	4.52	6.39	9.18	10.66	11.58	10.89	
Mal placés	0	1	4	28	167	2020	
Distance moyenne	0.00	0.00	0.01	0.03	0.11	1.05	
Distance maximale	0	1	2	2	4	32	

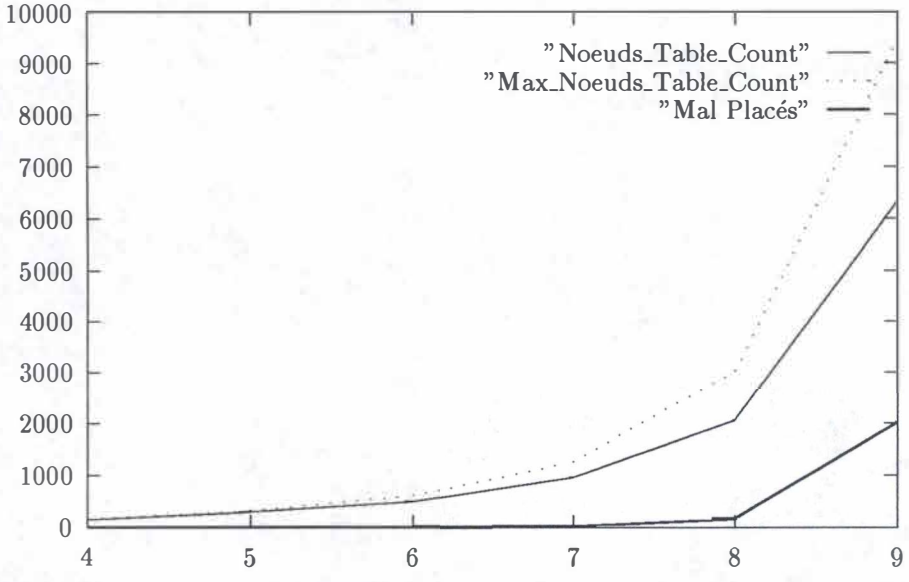
Comme l'on pouvait s'y attendre, le temps d'exécution croit exponentiellement. Le temps semble presque doubler quand l'échiquier hérite d'une ligne et d'une colonne supplémentaires. La tendance va de même pour les nœuds utilisés. Le facteur de partage semble tourner idéalement autour de dix ou onze pourcents, ce qui justifie pleinement l'implémentation des partages de nœuds. Quant aux chiffres des trois dernières lignes, ils suivent également une tendance croissante. Jusqu'ici, rien de bien extraordinaire.

Voici un aperçu graphique du tableau :

[17] Cette valeur fait l'objet d'un tableau propre



Quant aux nombre de nœuds utilisés, on obtient l'allure suivante :

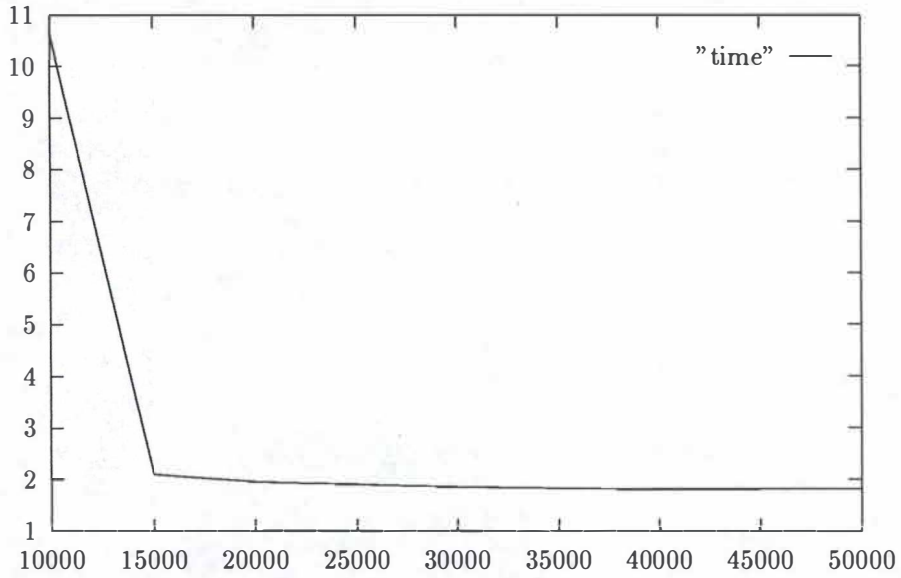


Toutefois, pour le cas $n = 9$, on regardera attentivement le paramètre *Max_Nœuds_Table_Count*. A un instant donné de l'exécution, le système nécessitait bien 9480 nœuds, montrant que le tabelau de hashing se remplit bel et bien. Pour $n = 10$, une table de hashing prévue pour 20000 nœuds ne tient plus la route : elle se sature et arrête l'exécution[18]. Les cas $n = 9$ et $n = 10$ font l'objet chacun d'un tableau propre où la taille de la table de hashing change.
Pour $n = 9$:

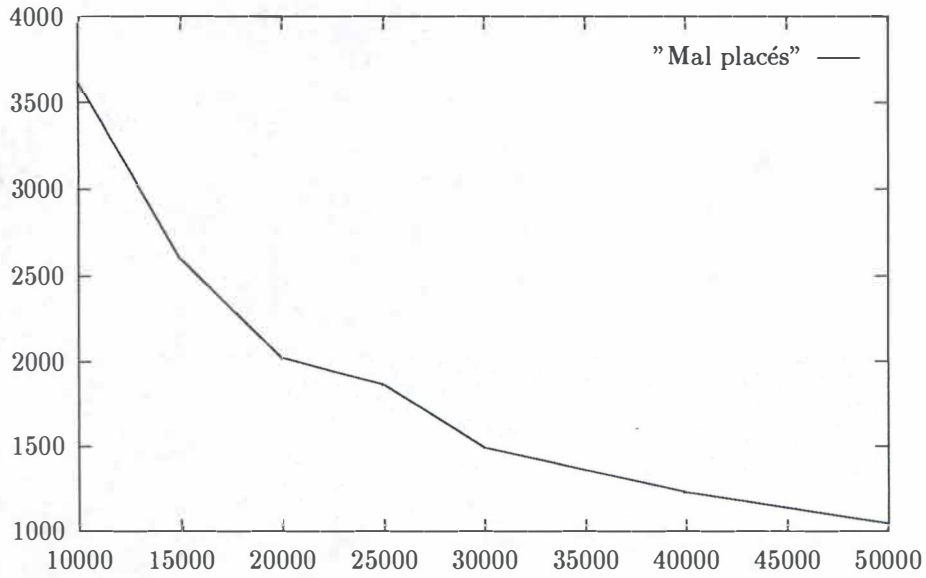
MAX_NOEUDS	10000	15000	20000	25000	30000	40000	50000
Temps	10.65	2.10	1.95	1.90	1.85	1.80	1.82
Nœuds_Table_Count	6326	6326	6326	6326	6326	6326	6326
Max_Nœuds_Table_Count	9480	9480	9480	9480	9480	9480	9480
Facteur de partage	10.89	10.89	10.89	10.89	10.89	10.89	10.89
Mal placés	3618	2593	2020	1862	1488	1231	1045
Distance moyenne	7.55	1.80	1.05	0.93	0.61	0.45	0.39
Distance maximale	704	43	32	25	16	14	13

[18] L'interpréteur ne permet pas encore de dilatation dynamique du tableau de hashing. Nous n'avons pas réfléchi au coût d'une telle opération sur l'exécution du programme.

Comprenez donc bien que le vitesse de résolution peut changer du tout au tout :



Ce gain de temps s'explique partiellement par le fait que le nombre de nœuds mal placés diminue, évitant ainsi des recherches inutiles dans la table de hashing :



Pour $n = 10$:

MAX_NOEUDS	20000	30000	40000	50000	60000	70000	80000	90000	100000	200000
Temps	95.70	530.33	31.66	11.64	8.84	7.35	6.81	6.50	6.40	6.15
Nœuds_Table_Count	24080	24080	24080	24080	24080	24080	24080	24080	24080	24080
Max_Nœuds_Table_Count	35549	35549	35549	35549	35549	35549	35549	35549	35549	35549
Facteur de partage	11.13	11.13	11.13	11.13	11.13	11.13	11.13	11.13	11.13	11.13
Mal placés			14034	12198	10521	9184	7878	7525	7031	4361
Distance moyenne			8.76	6.69	3.20	2.12	1.34	1.26	1.12	0.69
Distance maximale			846	358	200	127	41	44	45	25

Ces deux cas révèlent toute l'importance du parmétrage correct de la table de hashing. Ainsi, une table trop petite étouffe trop les nœuds parce que l'adresse calculée par la fonction de hashing risque trop souvent d'être déjà occupée par un autre nœuds. Avec $n = 10$, les cas où MAX_NOEUDS vaut 20000

et 30000 s'arrêttent car le table est trop petite, mais ils prennent aussi énormément plus de temps (jusqu'à 530 secondes, soit 8 minutes et 50 secondes !) parce que l'interpréteur passe le plus clair de son temps à chercher des places libres pour les nœuds. Ces nœuds mal placés sont également très nombreux. Nous n'assomerons plus le lecteur de graphiques assomants car les chiffres sont ici très tranchants.

L'utilisateur doit donc prendre conscience du pouvoir que lui donne le paramètre -t table_size à l'invocation. En ajustant ce paramètre judicieusement, il peut arriver à faire chuter le temps d'exécution d'un facteur trois ! (voir les colonnes de $n = 10$ où MAX_NOEUDS vaut 40000 et 50000).

En synthèse, nous avons montré qu'au moins deux paramètres gouvernaient le comportement de l'interpréteur :

- la taille du problème ;

- la taille de la table de hashing, même si elle est suffisamment grande.

23.4 Remarque sur la formulation d'un problème

Nous avons expliqué précédemment, que la langage PROLOG posait certaines contraintes vis-à-vis de la rédaction d'un programme. Le β-langage en pose également, mais d'une façon un peu différente. Pour rappel, les deux programme PROLOG suivants ne donnaient pas du tout les même résultats car le second bouclait :

```
Pere(a,b).
Pere(b,c).
Pere(a,d).
Ancetre(X,Y) :- Pere(X,Y).
Ancetre(X,Y) :- Pere(X,Z), Ancetre(Z,Y).
```

```
Pere(a,b).
Pere(b,c).
Pere(a,d).
Ancetre(X,Y) :- Pere(X,Y).
Ancetre(X,Y) :- Ancetre(Z,Y), Pere(X,Z).
```

Notre β-langage n'accuse pas cet inconvénient mais une formulation différente peut apporter des résulats différents, non pas dans la solution du problème comme c'est le cas de PROLOG, mais dans l'exécution de l'interpréteur. Dans cet optique, le problème des n reines fait l'objet de deux formulations différentes en β-langage pour $n = 8$. Le prédicat sol(C1,C2,C3,C4,C5,C6,c7,C8) a été refomulé en permuttant les appels qui le définissent :

```
sol(C1,C2,C3,C4,C5,C6,C7,C8)   :=
      not prise(a[1],C1,a[2],C2)   &   not prise(a[1],C1,a[3],C3) &
      not prise(a[1],C1,a[4],C4)   &   not prise(a[1],C1,a[5],C5) &
      not prise(a[1],C1,a[6],C6)   &   not prise(a[1],C1,a[7],C7) &
      not prise(a[1],C1,a[8],C8)   &

      not prise(a[2],C2,a[3],C3)   &   not prise(a[2],C2,a[4],C4) &
      not prise(a[2],C2,a[5],C5)   &   not prise(a[2],C2,a[6],C6) &
      not prise(a[2],C2,a[7],C7)   &   not prise(a[2],C2,a[8],C8) &

      not prise(a[3],C3,a[4],C4)   &   not prise(a[3],C3,a[5],C5) &
      not prise(a[3],C3,a[6],C6)   &   not prise(a[3],C3,a[7],C7) &
      not prise(a[3],C3,a[8],C8)   &

      not prise(a[4],C4,a[5],C5)   &   not prise(a[4],C4,a[6],C6) &
      not prise(a[4],C4,a[7],C7)   &   not prise(a[4],C4,a[8],C8) &

      not prise(a[5],C5,a[6],C6)   &   not prise(a[5],C5,a[7],C7) &
      not prise(a[5],C5,a[8],C8)   &

      not prise(a[6],C6,a[7],C7)   &   not prise(a[6],C6,a[8],C8) &

      not prise(a[7],C7,a[8],C8)   ;
```

Les résultats de cet essai furent les suivants :

	$n = 8$ (original)	$n = 8$ (modifié)
Temps	0.70	1.42
Nombre de solutions	92	92
MAX_NOEUDS	20000	20000
Nœuds_Table_Count	2070	3213
Max_Nœuds_Table_Count	3024	4202
Facteur de partage	11.58	202.93
Mal placés	167	334
Distance moyenne	0.11	0.14
Distance maximale	4	7

Etant donné qu'il est très difficile de formaliser ou rationnaliser les règles permettant d'écrire un programme, nous laissons ces chiffres aux observations du lecteur. L'important est qu'il comprenne que le β-langage, de par son approche par point fixe, ne butte pas sur les mêmes pierres d'achoppement que PROLOG, mais qu'une formulation différente d'un problème influe sur l'exécution. En synthèse, on peut dire que le β-langage rend la solution d'un problème indépendante de sa formulation, mais que l'exécution en dépend bel et bien.

23.5 Le choix d'une fonction de hashing

La fonction mathématique utilisée par la technique du hashing fait l'objet de beaucoup de précautions. Le choix conserve toutefois une part d'arbitraire et la fonction mathématique, à défaut d'être la plus optimale, est souvent la plus satisfaisante. Deux tendances opposées gouvernent la recherche

- complexifier la fonction de sorte que deux éléments distincts soient rangés dans deux cases distinctes, évitant ainsi de courir après un élément dans le tableau de hashing ;
- diminuer le nombre d'opérations effectuées par la fonction car de nombreux appels à cette fonction, combinés à un haut degré de complexité, risquent de ralentir l'entreprise global du programme.

Afin de montrer la difficulté de ce choix, nous allons considérer plusieurs fonctions[19] :

1.

$$H_1 = \left[v + \prod_{i=1}^{n} \log_{10}(fils_i) \right] \ mod\ (MAX\_NOEUDS - 3) \ + \ 2$$

2.
$$H_2 = \left[v + \sum_{i=1}^{n} \sqrt{fils_i} \right] \ mod\ (MAX\_NOEUDS - 3) \ + \ 2$$

3.
$$H_3 = \left[v + \prod_{i=1}^{n} fils_i \right] \ mod\ (MAX\_NOEUDS - 3) \ + \ 2$$

4.
$$H_4 = \left[v + \sum_{i=1}^{n} \log_{10}(fils_i \cdot 2^i) \right] \ mod\ (MAX\_NOEUDS - 3) \ + \ 2$$

5.
$$H_5 = \left[v + \sum_{i=1}^{n} fils_i \right] \ mod\ (MAX\_NOEUDS - 3) \ + \ 2$$

6.
$$H_6 = \left[v + \sum_{i=1}^{n} (MAX\_NOEUDS - fils_i) \right] \ mod\ (MAX\_NOEUDS - 3) \ + \ 2$$

7.
$$H_7 = \left[\sum_{i=1}^{n} fils_i^2 \right] \ mod\ (MAX\_NOEUDS - 3) \ + \ 2$$

8.
$$H_8 = \left[v + \sum_{i=1}^{n} fils_i^2 \right] \ mod\ (MAX\_NOEUDS - 3) \ + \ 2$$

9.
$$H_9 = \left[(218.959.117 \cdot (1973 \cdot v + \sum_{i=1}^{n} fils_i \cdot 2^i)) \ div\ 97 \right] \ mod\ (MAX\_NOEUD - 3) \ + \ 2$$

10.
$$H_{10} = \left[\sum_{i=1}^{n} (fils_i \cdot 2^{i+v} \right] \ mod\ (MAX\_NOEUD - 3) \ + \ 2$$

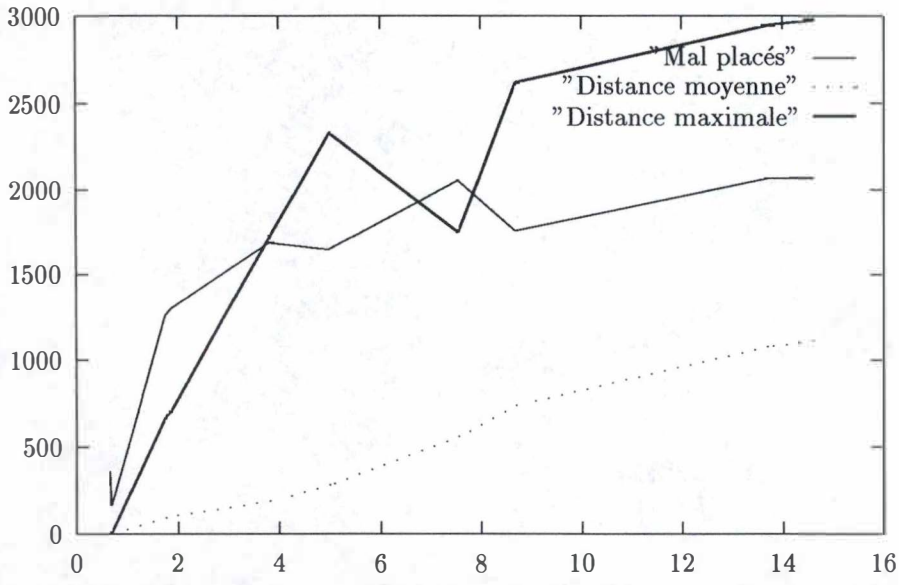
Nous avons essayé ces fonctions sur le problème des huit reines. Les résultats sont très différents d'une fonction à l'autre. Volontairement, nous avons déclaré la liste de fonctions par ordre décroissant selon le temps d'exécution. Voici le tableau synoptique où, implicitement, les paramètres *Max_Noeuds*, *Noeuds_Table_Count* et *Max_Noeuds_Table_Count* ont respectivement les valeurs 20000, 2070 et 3024 quelle que soit la fonction de hashing.

Fonction n°	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n = 7$	$n = 8$	$n = 9$[20]	$n = 10$
Temps	14.60	13.70	8.70	7.56	5.00	3.82	1.85	1.76	0.70	0.67
Facteur de partage	11.58	11.58	11.58	11.58	11.58	11.58	11.58	11.58	11.58	11.58
Mal placés	2068	2066	1757	2053	1646	1686	1293	1265	167	364
Distance moyenne	1112.82	1080.06	736.38	558.83	281.20	188.68	102.97	94.26	0.11	0.24
Distance maximale	2979	2944	2617	1751	2329	1721	699	676	4	6

Les trois dernières lignes ont été rangées dans le graphique suivant avec le temps pour axe d'abscisse :

[19] Etant donné que le tableau est de taille MAX_NOEUDS, mais que les deux premiers éléments sont toujours occupés par les relations *TRUE* et *FALSE*, il faut procéder à une recherche du reste de la division entière par $MAX\_NOEUDS - 3$ puis augmenter de deux unités.

[20] Cette fonction est celle qui a été choisie pour les tests expliqués plus tôt.



Ces résultats ne contredisent pas l'intuition que l'on peut avoir. Il est clair que les fonctions qui ne mettent pas de relief dans les informations d'un nœud ne seront pas performantes. Pour mieux cerner ce que cela signifie, considérons les fonctions H_5 et H_{10}. La fonction H_5 procède par une somme la plus plate possible, dans le sens où le même arbre mais avec deux fils permutés entre eux donnera le même résultat, et provoquera plus souvent une surchage d'une même adresse de base car cette fonction de hashing poussera l'interpréteur à mettre plusieurs nœuds distincts dans une case. La fonction H_{10} multiplie l'adresse d'un fils par 2^{i+v}, ce qui brise la linéarité dont faisait preuve la fonction H_5. Par la ligne résumant les distances moyennes, on perçoit donc le lien direct entre le temps d'exécution et l'écart dû à un mauvais placement par la fonction de hashing (la dépendance serait semble-t-il linéaire, mais n'en jurons rien car nous ne testons que quelques fonctions).
Les paramètres comme *Max_Noeuds*, *Noeuds_Table_Count* et *Max_Noeuds_Table_Count* sont resté les mêmes, ainsi que le facteur de partage. Cette observation est tout à fait logique puisque les notions qu'ils recouvrent sont indépendantes de la fonction de hashing utilisée. Ceci souligne bien, que dans ce contexte, deux niveaux coopèrent

- un niveau logique qui considère le contenu des nœuds, *i.e.* l'information ;
- un niveau pratique qui considère leurs stockage et manipulation.

Etant donné que le convention de partage des nœuds est basée sur l'information qu'ils renferment et non sur leur stockage, on comprend maintenant mieux cette indépendance vis-à-vis de la fonction de hashing. Une mesure du nombre d'appels à la fonction de hashing montre que, dans le cas de base (fonction de hashing H_9, que nous avons choisie pour nos tests), l'interpréteur procède à 9591 appels. Ainsi, sur les 0.70 secondes, 0.19 sont consacrées à du calcul de hashing. Dans le cas des neufs reines, on passe à 235600 appels utilisant 0.47 secondes sur les 1.95. Dans le cas des dix reines, on passe à 754296 appels utilisant 1.48 secondes sur les 11.68. La durée de chaque calcul est donc de $1.98e^{-06}$ seconde.
Cette observation rejoint le fait que l'on ne peut se permettre de trop complexifier la fonction de hashing sans être pénalisé par des calculs plus longs. On peut alors regarder les fonctions H_9 et H_{10}. La fonction H_9 est codée en C de la façon suivante :

```
int H_9(int v, int n, int* fils){
  unsigned int adr, d;
  int i;

  adr = v*1973;
  for(i = 0; i < n; i++) adr += fils[i]<<i;
  d = adr << 8;
  adr += d;
  d = d << 8;
  adr += d;
  d = d << 8;
```

```
  adr += d;
  adr *= 13;
  adr /= 97;
  adr %= (MAX_NOEUDS - 3);
  adr += 2;
  return adr;
}
```

La fonction H_{10} est plus simple

```
int FH_9(int v, int n, int* fils){
  unsigned int adr;
  int i;

  adr=0;
  for(i = 0; i < n; i++) adr += (fils[i]<<((i+v)));
  adr= (adr % (MAX_NOEUDS-3))+2;
  return adr;
}
```

Toutefois, bien que la fonction H_9 mette mieux en relief les informations du nœuds, H_{10} la coiffe sur la ligne. La simplicité l'a ici emporter ; la fonction H_9 battait pourtant le fonction H_{10} en termes de nœuds mieux placés, de distances moyennes et maximales de l'adresse de base plus courtes.

23.6 Suggestion d'application

En disposant d'un interface plus conviviale que la ligne de commande, on peut imaginer un logiciel de recherche d'horaires dans les écoles.

Quatrième partie
Application à l'interprétation abstraite

Dans cette partie, nous allons décrire comment notre interpréteur peut être utilisé pour l'interprétation abstraite de programmes PROLOG.

24 Une sémantique concrète pour PROLOG

Dans la partie préliminaire de ce travail, nous avons déjà (de façon intuitive) introduit une sémantique pour PROLOG. Il s'agissait là de la sémantique opérationnelle. Il existe encore d'autres formulations de cette sémantique qui sont parfois plus utile dans un contexte donné. Ainsi on a une formulation en termes des modèles de Herbrand et une formulation dénotationelle. C'est cette dernière que nous utiliserons dans cette partie.

Notre objectif étant d'écrire un texte compréhensible pour le plus de monde possible, nous allons consacrer ce chapitre à une présentation de cette sémantique. D'autres présentations peuvent bien entendu être trouvées dans la littérature spécialisée.

24.1 La syntaxe abstraite

Dans le reste de cette partie, un programme PROLOG sera considéré comme suit :

- Un programme est une suite de procédures :

  $$\text{prog} ::= \text{proc}_1; \ldots; \text{proc}_n$$

- Une procédure est composée d'un nom, d'une arité et d'une suite de goals :

  $$\text{proc} ::= p(X_1, \ldots, X_n) := g_1 \mid \ldots \mid g_k$$

- Un goal est une suite d'atomes :

  $$g ::= a_1, \ldots, a_n$$

- Un atome est soit
  - une unification de variables :

    $$X_i = X_j$$

  - une unification fonctionnelle :

    $$X_{i_1} = f(X_{i_2}, \ldots, X_{i_p})$$

  - un appel de procédure :

    $$p(X_{i_1}, \ldots, X_{i_n}).$$

24.2 Les substitutions

La matière première de tout interpréteur PROLOG consiste bien évidemment des substitutions, nous les définissons comme suit :

Définition 29 (Substitution)
Une substitution est une expression de la forme

$$\{X_1/t_1, \ldots, X_n/t_n\},$$

où les t_i sont des termes dont les variables n'apparaissent pas dans l'ensemble

$$\{X_1, X_2, X_3, \ldots\}.$$

Le fait que les variables dans les termes t_i ne puissent pas être des X_j n'est pas une restriction puisque ceci peut toujours être obtenu en appliquant une substitution inversible. Nous avons ajouté cette condition seulement parce qu'elle simplifie un peu le raisonnement pour la dérivation de la sémantique abstraite.

24.3 Les environnements

Définition 30 (Environnement)
Soit P un programme PROLOG, $\{p_1, \ldots, p_\nu\}$ les prédicats de P.

Un environnement pour P est une application

$$\{p_1, \ldots, p_\nu\} \to \mathcal{S}ubst \to \mathcal{P}(\mathcal{S}ubst).$$

Nous notons l'ensemble de ces environnements par $\mathcal{E}nv$.

Un environnement décrit en fait comment chaque prédicat d'un programme PROLOG va transformer une substitution d'entrée à un ensemble de substitutions de sortie.

De cette sorte, ces environnements sont bien appropriés pour décrire la sémantique d'un programme PROLOG.

Si on considère par exemple le programme ci-dessous :

$$\begin{cases} \text{natPlus}(X_1, X_2, X_3) & :- \quad X_1 = 0, X_2 = X_3. \\ \text{natPlus}(X_1, X_2, X_3) & :- \quad X_1 = s(X_4), X_3 = s(X_5), \text{natPlus}(X_4, X_2, X_5). \end{cases}$$

la réponse à la requête

$$? \ \text{natPlus}(s(s(0)), s(0), Y_1)$$

sera

$$\{Y_1/s(s(s(0)))\},$$

ce qul'on peut interpréter comme si le prédicat *natPlus* transforme la substitution d'entrée

$$\{X_1/s(s(0)), X_2/s(0), X_3/Y_1\}$$

en la substitution

$$\{X_1/s(s(0)), X_2/s(0), X_3/s(s(s(0)))\}.$$

La sémantique de P sera donc un environnement *env* tel que :

$$(\text{env natPlus}) \ \{X_1/s(s(0)), X_2/s(0), X_3/Y_1\} = \{\{X_1/s(s(0)), X_2/s(0), X_3/s(s(s(0)))\}\}.$$

De la même façon, on s'attend à trouver les transformations suivantes dans la fonction *(env natPlus)* :

$$\begin{cases} \{X_1/Y_1, X_2/Y_2, X_3/s(s(0))\} & \mapsto & \{X_1/0, X_2/s(s(0)), X_3/s(s(0))\} \\ & & \{X_1/s(0), X_2/s(0), X_3/s(s(0))\} \\ & & \{X_1/s(s(0)), X_2/0, X_3/s(s(0))\} \\ \{X_1/Y_1, X_2/Y_1, X_3/s(s(0))\} & \mapsto & \{X_1/s(0), X_2/s(0), X_3/s(s(0))\} \\ \{X_1/s(0), X_2/Y_1, X_3/Y_2\} & \mapsto & \{X_1/s(0), X_2/Y_3, X_3/s(Y_3)\} \end{cases}$$

24.4 La sémantique d'une sous-formule

Avant que l'on puisse spécifier quel environnement sera considéré comme la sémantique du programme, on doit d'abord décrire la sémantique des sous-formules.

La sémantique d'une sous-formule f est une fonction

$$\begin{array}{rlllll} \mathcal{F}\,[\![f]\!] & : & \mathcal{E}nv & \rightarrow & \mathcal{S}ubst & \rightarrow \quad \mathcal{P}(\mathcal{S}ubst) \\ & : & \text{env} & \rightarrow & \sigma & \rightarrow \quad \mathcal{F}\,[\![f]\!]\ \text{env}\ \sigma \end{array}$$

qui étant donné une connaissance partielle sur les prédicats du programme (c'est-à-dire un environnement) va associer à chaque substitution, un ensemble de solutions.

Cette fonction $\mathcal{F}\,[\![f]\!]$ peut être définie comme suit :

- Pour une unification de variables :

$$\mathcal{F}\,[\![X_i = X_j]\!]\ \text{env}\ \{X_1/t_1, \ldots, X_n/t_n\} = \{\{X_1/t_1\sigma, \ldots, X_n/t_n\sigma\}\},$$

où $\sigma = \text{mgu}(t_i, t_j)$ est l'unificateur le plus général de t_i et t_j.

- Pour une unification fonctionelle :

$$\mathcal{F}\,[\![X_{i_1} = f(X_{i_2}, \ldots, X_{i_p})]\!]\ \text{env}\ \{X_1/t_1, \ldots, X_n/t_n\} = \{\{X_1/t_1\sigma, \ldots, X_n/t_n\sigma\}\},$$

où $\sigma = \text{mgu}(t_{i_1}, f(t_{i_2}, \ldots, t_{i_p}))$.

- Pour un appel de procédure :

$$\{X_1/t_1\sigma, \ldots, X_n/t_n\sigma\} \in \mathcal{F}\,[\![p(X_{i_1}, \ldots, X_{i_k})]\!]\ \text{env}\ \{X_1/t_1, \ldots, X_n/t_n\}$$

$$\Leftrightarrow \exists t'_1, \ldots, t'_k \quad : \quad \{X_1/t'_1, \ldots, X_k/t'_k\} \in (\text{env}\ p)(\{X_1/t_{i_1}, \ldots, X_k/t_{i_k}\})$$

et σ est la substitution la plus générale telle que :

$$\begin{cases} t'_1 = t_{i_1}\sigma \\ \vdots \\ t'_k = t_{i_k}\sigma \end{cases}$$

- Pour un goal :

$$\mathcal{F}\,[\![\alpha_1, \ldots, \alpha_n]\!]\ \text{env}\ \theta = \bigcup_{\theta' \in \mathcal{F}[\![\alpha_1]\!]\ \text{env}\ \theta} \mathcal{F}\,[\![\alpha_2, \ldots, \alpha_n]\!]\ \text{env}\ \theta'$$

- Pour une procédure :
  - $\mathcal{F}\,[\![p(X_1, \ldots, X_k) :- g]\!]\ \text{env}\ \{X_1/t_1, \ldots, X_k/t_k\}$
    $= \mathcal{F}\,[\![g]\!]\ \text{env}\ \{X_1/t_1, \ldots, X_k/t_k, X_{k+1}/t_{k+1}, \ldots, X_{k+l}/t_{k+l}\}$

    où $X_1, \ldots, X_k, X_{k+1}, \ldots, X_{k+l}$ sont toutes les variables qui apparaissent dans g et où $t_{k+1}, \ldots, t_{k+l}$ sont de nouvelles variables.
  - $\mathcal{F}\,[\![p(X_1, \ldots, X_k) :- g_1 \mid \ldots \mid g_n]\!]\ \text{env}\ \theta$
    $= \bigcup_{i=1}^{n} \mathcal{F}\,[\![p(X_1, \ldots, X_k) :- g_i]\!]\ \text{env}\ \theta$

24.5 La sémantique d'un programme

Muni de l'opérateur $\mathcal{F}$ qui décrit la sémantique des sous-formules, on peut maintenant définir un opérateur

$$\Xi : \mathcal{E}nv \to \mathcal{E}nv$$

qui prend une approximation de la sémantique de P et en renvoie une nouvelle comme suit :

$$(\Xi\ \text{env})\ p_i\ \theta = \mathcal{F}\,[\![proc]\!]\ \text{env}\ \theta$$

où *proc* est la procédure qui correspond au prédicat p_i.

La sémantique de P est alors définie comme étant le plus petit point fixe de cet opérateur.

25 Une sémantique abstraite pour PROLOG

Comme nous l'avons vu au chapitre précédent, la sémantique d'un programme PROLOG va associer à chaque prédicat du programme, une fonction

$$\mathcal{S}ubst \rightarrow \mathcal{P}(\mathcal{S}ubst)$$

qui décrit comment le prédicat transforme une substitution d'entrée en un ensemble de substitutions de sortie.

Le but de l'interprétation abstraite est d'essayer d'obtenir des informations partielles sur ces fonctions de transformation.

Une telle information partielle est, par exemple, que le prédicat *natPlus* considéré à la page (138) tranforme chaque substitution d'entrée

$$\{X_1/t_1, X_2/t_2, X_3/t_3\}$$

pour laquelle t_1 et t_2 sont des termes clos et le terme t_3 est une variable, en une substitution de sortie pour laquelle les trois termes sont clos.

L'approche suivie en Interprétation Abstraite est très rigoureuse :

- On définit un domaine abstrait $\mathcal{S}ubst^*$, c'est-à-dire un ensemble dont chaque élément représente tout un ensemble de substitutions.
- Ensuite, on essaye de transposer la sémantique concrète qui est définie en termes de substitutions simples à une sémantique qui est définie en les termes de ces ensembles de substitutions.
- Si le domaine abstrait et l'abstraction de la sémantique sont bien choisis, on peut alors essayer de calculer cette sémantique abstraite tout en détail.

25.1 Le domaine abstrait

Comme domaine abstrait, on va considérer l'ensemble $\mathcal{S}ubst^*$ suivant :

$$\mathcal{S}ubst^* = \bigoplus_{n=0}^{\infty} \{\{X_1/m_1, \ldots, X_n/m_n\} \mid m_1, \ldots, m_n \in \mathcal{M}\},$$

où

$$\mathcal{M} = \{g, f, v, a\}.$$

Chaque élément de $\mathcal{S}ubst^*$ correspond à un ensemble de substitutions si on considère la fonction (dite de concrétisation) suivante :

$$\mathcal{C}c : \mathcal{S}ubst^* \rightarrow \mathcal{P}(\mathcal{S}ubst^*)$$

définie par :

$\{X_1/t_1, \ldots, X_n/t_n\} \in \mathcal{C}c(\{X_1/m_1, \ldots, X_n/m_n\})$
ssi pour chaque $i \in \{1, \ldots, n\}$

- $m_i = g \Rightarrow t_i$ est clos
- $m_i = v \Rightarrow t_i$ est une variable
- $m_i = f \Rightarrow t_i$ est une variable et
  $t_i \notin \mathrm{var}(\{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n\})$.

C'est à cause de cette correspondance qu'on appelera les éléments de $\mathcal{S}ubst^*$ des substitutions abstraites.

25.2 Les environnements abstraits

Définition 31 (Environnement abstrait)
Soit P un programme PROLOG, $\{p_1, \ldots, p_\nu\}$ *les prédicats de P.*

Un environnement abstrait pour P est une application

$$\{p_1, \ldots, p_\nu\} \to \mathcal{S}ubst^* \to \mathcal{P}(\mathcal{S}ubst^*).$$

Nous notons l'ensemble de ces environnements abstraits par $\mathcal{E}nv^*$.

Tout comme une substitution abstraite représente un ensemble de substitutions concrètes, on peut considérer une environnement abstrait comme un ensemble d'environnements concrets.

De façon rigoureuse, ceci peut être exprimé à l'aide de la fonction

$$\mathcal{C}c' : \mathcal{E}nv^* \to \mathcal{P}(\mathcal{E}nv)$$

définie par :

$$\mathcal{C}c'(\text{env}^*) = \left\{ \text{env} \;\middle|\; \forall p, \theta, \theta^* : \theta \in \mathcal{C}c(\theta^*) \Rightarrow (\text{env } p)\theta \subseteq \bigcup_{\tau^* \in (\text{env}^* \; p)(\theta^*)} \mathcal{C}c(\tau^*) \right\}$$

Cette définition fait un peu mal aux yeux, pourtant elle est très intuitive. Elle ne fait qu'exprimer qu'une expression telle que

$$(\text{env}^* \; p)\{X_1/g, X_2/g, X_3/a\} = \{\{X_1/g, X_2/g, X_3/g\}, \{X_1/g, X_2/g, X_3/f\}\},$$

correspond au niveau concret à la contrainte que les fonctions (env p) correspondantes, vont transformer chaque substitution de type

$$\{X_1/g, X_2/g, X_3/a\}$$

en une substution de type

$$\{X_1/g, X_2/g, X_3/g\} \text{ ou } \{X_1/g, X_2/g, X_3/f\}.$$

25.3 La sémantique abstraite d'une sous-formule

Rappelons que la sémantique concrète d'une sous-formule f a été définie comme une fonction

$$\mathcal{F}\ |[f]|: \mathcal{E}nv \to \mathcal{S}ubst \to \mathcal{P}(\mathcal{S}ubst).$$

Dans cette section, nous allons définir la sémantique abstraite de f comme étant une fonction

$$\mathcal{F}^*\ |[f]|: \mathcal{E}nv^* \to \mathcal{S}ubst^* \to \mathcal{P}(\mathcal{S}ubst^*).$$

Il est clair qu'une certaine relation devra exister entre $\mathcal{F}\ |[f]|$ et $\mathcal{F}^*\ |[f]|$ pour que la sémantique abstraite d'un programme soit effectivement une abstraction de la sémantique concrète.

La relation nécessaire est celle de la sûreté :

On veut que

$$\mathcal{F}^*\ |[f]|\ \text{env}^*\ \sigma^* = \{\tau_1^*, \ldots, \tau_n^*\},$$

$$\sigma \in \mathcal{C}c(\sigma^*) \text{ et } \text{env} \in \mathcal{C}c'(\text{env}^*)$$

implique que :

$$\mathcal{F}\ |[f]|\ \text{env}\ \sigma \subseteq \bigcup_{i=1}^{n} \mathcal{C}c(\tau_i^*).$$

Nous pouvons maintenant procéder à la définition de la sémantique abstraite pour chaque type de sous-formule :

25.3.1 La sémantique abstraite d'une unification de variables

Considérons pour chaque $n \in \mathbb{N}_0$, la fonction

$$\begin{aligned} \text{unif\_var}_n \quad &: \mathcal{M}^n \to \mathcal{P}(\mathcal{M}^n) \\ &: < m_1, \ldots, m_n > \to \{< m_1', \ldots, m_n' >\} \end{aligned}$$

définie par le tableau suivant :

m_1	m_2	m'_1, m'_2	$m'_3, \ldots, m'_n$
g	g	g	–
g	f	g	–
g	v	g	$v \rightarrow g/v$
g	a	g	$v \rightarrow g/v$
f	f	v	–
f	v	v	–
f	a	a	–
v	v	v	–
v	a	a	$v \rightarrow a$
a	a	–	$v \rightarrow a$

et par symétrie :

$$\text{unif\_var}_n(m_1, m_2, \ldots, m_n) = \text{unif\_var}_n(m_2, m_1, \ldots, m_n).$$

Ce tableau doit être lu de la façon suivante :

- La partie gauche du tableau définit les différents cas à considérer
- La partie droite définit par colonne quelques composants de l'image, en l'occurence on considère les colonnes m'_1, m'_2 et $m'_3, \ldots, m'_n$.
  - La signification d'un g, f, v ou a dans une de ces colonnes est claire. Elle veut dire que les paramètres concernés prennent cette valeur.
  - Un tiret dans une de ces colonnes veut dire que le paramètre de sortie concerné est identique au paramètre d'entrée qui y correspond.
    Ainsi un tiret dans la colonne m'_1, m'_2 veut dire que $m'_1 = m_1$ et $m'_2 = m_2$.
  - Une expression comme $v \rightarrow g/v$ dans une de ces colonnes veut dire que
    - si le paramètre d'entrée correspondant est v, le paramètre de sortie peut prendre une des deux valeurs g ou v
    - dans le cas contraire, le paramètre de sortie concerné est identique à son paramètre d'entrée.

Ainsi, la troisième ligne du tableau doit être lue comme suit :

$$\begin{aligned} & m_1 = g \wedge m_2 = v \\ & \quad \Rightarrow m'_1 = m'_2 = g \wedge \forall i \in \{3, \ldots, n\} : \{[m_i = v \Rightarrow m'_i \in \{g, v\}] \wedge [m_i \neq v \Rightarrow m'_i = m_i]\} \end{aligned}$$

La sémantique d'une unification de variables peut maintenant être définie à l'aide de cette fonction :

$$\begin{aligned} & \mathcal{F}^* \, [\![X_i = X_j]\!] \;\; \text{env}^* \; \{X_1/m_1, \ldots, X_n/m_n\} \ni \{X_1/m'_1, \ldots, X_n/m'_n\} \\ \Leftrightarrow \\ & \text{unif\_var}_n(m_i, m_j, m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_{j-1}, m_{j+1}, \ldots, m_n) \\ & \quad \ni (m'_i, m'_j, m'_1, \ldots, m'_{i-1}, m'_{i+1}, \ldots, m'_{j-1}, m'_{j+1}, \ldots, m'_n) \end{aligned}$$

Sûreté

Pour démontrer qu'il s'agît ici d'une abstraction sûre, on peut (par symétrie) se limiter au cas où $i = 1$ et $j = 2$.

On doit alors démontrer l'assertion suivante :

Soit
$\theta^* = \{X_1/m_1, \ldots, X_n/m_n\} \in \mathcal{S}ubst^*$ une substitution abstraite,
$\theta = \{X_1/t_1, \ldots, X_n/t_n\} \in \mathcal{C}c(\theta^*)$
et
$\sigma = \text{mgu}(t_1, t_2)$.

Alors il existe un tuple
$(m'_1, \ldots, m'_n) \in \text{unif\_var}_n(m_1, \ldots, m_n)$
tel que
$\theta\sigma = \{X_1/t_1\sigma, \ldots, X_n/t_n\sigma\} \in \mathcal{C}c(\{X_1/m'_1, \ldots, X_n/m'_n\})$.

Considérons pour la démonstration les différents cas qui peuvent se produire :

- **$\mathbf{m_1 = g \wedge m_2 = g}$**

  t_1 et t_2 sont ground,
  s'ils sont unifiable, on peut prendre $\sigma = \{\ \}$ comme mgu.
  Tous les termes restent invariant et comme la même chose vaut pour les modes, on a démontré l'assertion dans ce cas-ci.

- **$\mathbf{m_1 = g \wedge m_2 = f}$**

  Dans ce cas, $t_1 = c$ est un terme clos et $t_2 = Y$ est une variable qui n'apparaît pas dans les autres termes t_i.
  Comme mgu pour l'unification de c et Y on peut considérer la substitution $\{Y/c\}$.
  On a alors que :

  $$\theta\sigma = \{X_1/c, X_2/c, X_3/t_3\sigma, \ldots, X_n/t_n\sigma\}$$

  et comme Y n'apparaît pas dans $t_3, \ldots, t_n$:

  $$\theta\sigma = \{X_1/c, X_2/c, X_3/t_3, \ldots, X_n/t_n\}$$

  Les deux premiers termes sont ground et les autres ont toujours le même mode, on a donc effectivement que

  $$\theta\sigma \in \mathcal{C}c(\{X_1/g, X_2/g, X_3/m_3, \ldots, X_n/m_n\}).$$

- **$\mathbf{m_1 = g \wedge m_2 = v}$**

  Dans ce cas, t_1 est un terme clos c et t_2 est une variable Y qui peut apparaître dans les autres termes t_i.
  Comme mgu pour l'unification de c et Y on peut conidérer la substitution $\{Y/c\}$.
  On a alors que :

  $$\theta\sigma = \{X_1/c, X_2/c, X_3/t_3\sigma, \ldots, X_n/t_n\sigma\}$$

  Les deux premiers termes sont ground et pour les autres termes $t_i\sigma$ avec $i \in \{3, \ldots, n\}$ on a que :

  - Si $m_i = g$, t_i est ground et $t_i\sigma$ le sera aussi.
    Ceci correspond avec $m'_i = g$.
  - Si $m_i = f$, t_i est une variable qui n'apparaît pas dans le autres termes t_j et qui est donc en particulier différent de Y. Par conséquent $t_i\sigma = t_i$ est toujours une variable qui n'apparaît pas dans les autres termes.
    Ceci correspond avec $m'_i = f$.
  - Si $m_i = v$, t_i est une variable qui peut ou qui ne peut pas être égal à Y. On a donc que, soit $t_i\sigma = t_i$, soit $t_i\sigma = c$, donc tout ce qu'on peut dire est que $t_i\sigma$ est soit ground, soit une variable.
    Ceci correspond avec $m'_i = g$ ou $m'_i = v$.
  - Si $m_i = a$, t_i est un terme quelconque, et $t_i\sigma$ le sera aussi.
    Ceci correspond avec $m'_i = a$.

  On peut donc effectivement trouver un tuple $(m'_1, \ldots, m'_n)$ tel que

  $$\theta\sigma \in \mathcal{C}c(\{X_1/m'_1, \ldots, X_n/m'_n\}),$$

et

$$m'_1 = g \wedge m'_2 = g \text{ et}$$
$$\forall i \in \{3, \ldots, n\} : \{[m_i = v \Rightarrow m'_i \in \{g, v\}] \wedge [m_i \neq v \Rightarrow m'_i = m_i]\}$$

- $\mathbf{m_1 = g \wedge m_2 = a}$

  Dans ce cas, t_1 est un terme clos c et t_2 un terme quelconque.
  Le mgu pour l'unification de c et t_2 va substituer les variables dans t_2 pour des termes clos.
  Le reste de l'argument est identique au cas précédent.

- $\mathbf{m_1 = f \wedge m_2 = f}$

  Dans ce cas, t_1 (resp. t_2) est une variable Y_1 (resp. Y_2) qui n'apparaît pas dans d'autres termes.
  Comme mgu pour l'unification de Y_1 et Y_2 on peut considérer la substitution $\{Y_2/Y_1\}$.
  On a alors que :

$$\begin{aligned} \theta\sigma &= \{X_1/Y_1, X_2/Y_2, X_3/t_3\sigma, \ldots, X_n/t_n\sigma\} \\ &= \{X_1/Y_1, X_2/Y_1, X_3/t_3, \ldots, X_n/t_n\} \end{aligned}$$

  Les deux premiers termes sont devenus des variables qui apparaissent aussi dans un autre terme.
  Ceci correspond à $m'_1 = m'_2 = v$.

  Les autres termes ne sont pas affectés par la substitution σ.
  Ceci correspond à $m'_i = m_i$.
  On a donc effectivement que

$$\theta\sigma \in Cc(X_1/v, X_2/v, X_3/m_3, \ldots, X_n/m_n).$$

Les autres cas sont tout à fait similaire, on les décrit très sommairement :

- $\mathbf{m_1 = f \wedge m_2 = v}$

  t_1 est une variable non-partagée. L'effet de la substitution σ est de l'unifier avec la variable qu'est le terme t_2. Par conséquent on a que le premier terme devient une variable partagée et donc que $m'_1 = v$. Les autres termes ne ont pas affectés par la substition et peuvent être décrits par le même mode qu'avant.

- $\mathbf{m_1 = f \wedge m_2 = a}$

  t_1 est une variable non-partagée. L'effet de la substitution σ est de l'unifier avec le terme t_2. Par conséquent on a que le premier terme devient *any* et donc que $m'_1 = a$. Les autres termes ne ont pas affectés par la substition et peuvent être décrits par le même mode qu'avant.

- $\mathbf{m_1 = v \wedge m_2 = v}$

  t_1 et t_2 sont des variables, qui vont être unifiées par la substitution σ. Cette unification ne change rien au niveau des modes.

- $\mathbf{m_1 = v \wedge m_2 = a}$

  t_1 est une variable partagée que la substitution σ va substituer par le terme t_2. Le premier terme devient donc *any*.
  Les seules terme $t_3, \ldots, t_n$ qui peuvent être affectés par σ, sont ceux qui sont *any* ou les variables partagées.
  Les termes qui sont *any* peuvent toujours être décrits comme *any*.
  Les variables partagées restent invariantes si elles sont différentes de t_1 ou sont remplacées par t_2 dans le cas contraire, leur mode devient donc *any*.
  Ceci correspond à la substitution abstraite

$$\{X_1/a, X_2/a, X_3/m'_3, \ldots, X_n/m'_n\}$$

  où

$$\forall i \in \{3, \ldots, n\} : m'_i = (v \rightarrow a)(m_i).$$

- $\mathbf{m_1 = a \wedge m_2 = a}$

  La substitution θ peut contenir dans son domaine toutes les variables qui apparaissent dans t_1 ou t_2. Par conséquent chaque terme t'_i ($i \in \{3, \ldots, n\}$) qui est une variable partagée peut être affectée

par cette substitution et doit être décrit par le nouveau mode $m'_i = a$.
Ceci correspond à la substitution abstraite

$$\{X_1/a, X_2/a, X_3/m'_3, \ldots, X_n/m'_n\}$$

où

$$\forall i \in \{3, \ldots, n\} : m'_i = (v \to a)(m_i).$$

25.3.2 La sémantique abstraite d'une unification fonctionelle

Considérons pour chaque $p \in \mathbb{N}_0$ et $q \in \mathbb{N}$ la fonction

$$\begin{aligned} \text{unif\_func}_{p,q} \quad &: \mathcal{M}^{p+q} \to \mathcal{M}^{p+q} \\ &: (m_1, m_2, \ldots, m_p, m_{p+1}, \ldots, m_{p+q}) \mapsto (m'_1, m'_2, \ldots, m'_p, m'_{p+1}, \ldots, m'_{p+q}) \end{aligned}$$

définie par les trois tableaux ci-dessous :

1. si $\mathbf{m_2 = \ldots = m_p = g}$

m_1	m'_1	$m'_2, \ldots, m'_p$	$m'_{p+1}, \ldots, m'_{p+q}$
g	–	–	–
f	g	–	–
v, a	g	–	$v \to g/v$

2. si $\mathbf{m_2, \ldots, m_p \in \{g, f\} \wedge \exists i : (i \in \{2, \ldots, p\} \wedge m_i \neq g)}$

m_1	m'_1	$m'_2, \ldots, m'_p$	$m'_{p+1}, \ldots, m'_{p+q}$
g	–	g	–
f	a	–	–
v	a	–	$v \to a$
a	a	$v/f \to a$	$v \to a$

3. si $\mathbf{\exists i : (i \in \{2, \ldots, p\} \wedge m_i \in \{v, a\})}$

m_1	m'_1	$m'_2, \ldots, m'_p$	$m'_{p+1}, \ldots, m'_{p+q}$
g	–	g	$v \to g/v$
f	a	–	–
v	a	– ! ! !	$v \to a$
a	–	$v/f \to a$	$v \to a$

La sémantique abstraite d'une unification fonctionelle peut alors être définie comme suit :

$$\mathcal{F}^* \, [\![X_{i_1} = f(X_{i_2}, \ldots, X_{i_p})]\!] \text{ env}^* \, \{X_1/m_1, \ldots, X_{p+q}/m_{p+q}\} \ni \{X_1/m'_1, \ldots, X_{p+q}/m'_{p+q}\}$$

$\Leftrightarrow$

$$\begin{aligned} &\text{unif\_func}_{p,q}(m_{i_1}, m_{i_2}, \ldots, m_{i_p}, m_{j_1}, \ldots, m_{j_q}) \ni (m'_{i_1}, m'_{i_2}, \ldots, m'_{i_p}, m'_{j_1}, \ldots, m'_{j_q}) \\ &\quad \text{où} : \{X_{j_1}, \ldots, X_{j_q}\} = \{X_1, \ldots, X_n\} - \{X_{i_1}, X_{i_2}, \ldots, X_{i_p}\}, \end{aligned}$$

Sûreté

Pour démontrer qu'il s'agît ici d'une abstraction sûre, on peut (par symétrie) se limiter au cas où l'unification considérée a la forme :

$$X_1 = f(X_2, \ldots, X_p).$$

On doit alors démontrer l'assertion suivante :

Soit
$\theta^* = \{X_1/m_1, \ldots, X_{p+q}/m_{p+q}\} \in \mathcal{S}ubst^*$ une substitution abstraite,
$\theta = \{X_1/t_1, \ldots, X_{p+q}/t_{p+q}\} \in \mathcal{C}c(\theta^*)$
et
$\sigma = \text{mgu}(t_1, f(t_2, \ldots, t_p))$.

Alors il existe un tuple
$(m'_1, \ldots, m'_{p+q}) \in \text{unif\_func}_{p,q}(m_1, \ldots, m_{p+q})$
tel que
$\theta\sigma = \{X_1/t_1\sigma, \ldots, X_{p+q}/t_{p+q}\sigma\} \in \mathcal{C}c(\{X_1/m'_1, \ldots, X_{p+q}/t_{p+q}\})$.

Considérons à nouveau les différents cas qui peuvent se produire :

1. $\mathbf{m_2 = \ldots = m_p = g}$

   Dans ce cas, $f(t_2, \ldots, t_p)$ est ground.

   (a) $\mathbf{m_1 = g}$

   t_1 est ground et on peut prendre $\sigma = \{\}$.
   Aucun terme n'est modifié par σ et tous les modes restent valables.

   (b) $\mathbf{m_1 = f}$

   t_1 est une variable non-partagée, disons Y.
   Comme unificateur on peut considérer $\sigma = \{Y/f(t_2, \ldots, t_p)\}$.
   t_1 devient ground par cette substitution et comme la variable est non-partagée, aucun autre terme n'est affecté.

   (c) $\mathbf{m_1 = v, a}$

   La substitution σ va substituer les variables dans t_1 par des termes clos.

   - t_1 devient ground.
   - Les termes $t_2, \ldots, t_p$, étant ground, ne sont pas affectés.
   - Les termes $t_{p+1}, \ldots, t_{p+q}$ qui sont des variables partagées deviennent ground ou ne sont pas affectés, selon que oui ou non cette variable apparaît dans t_1. Ces termes doivent donc être décrits par le mode g ou v.

2. $\mathbf{m_2, \ldots, m_p \in \{g, f\} \wedge \exists i : (i \in \{2, \ldots, p\} \wedge m_i \neq g)}$

   (a) $\mathbf{m_1 = g}$

   σ va substituer les termes t_i dans $f(t_2, \ldots, t_p)$ qui sont des variables, par des termes clos.
   Les modes $m'_2, \ldots, m'_p$ deviennent ainsi *ground*.
   Et comme toutes ces variables sont non-partagées et n'apparaissent dès lors pas dans les termes $t_{p+1}, \ldots, t_{p+q}$, ces derniers ne sont pas affectés par la substitution.

   (b) $\mathbf{m_1 = f}$

   t_1 est une variable non-partagée, disons Y.
   Comme unificateur σ on peut considérer la substitution

   $$\sigma = \{Y/f(t_2, \ldots, t_p)\}.$$

   Comme la variable Y apparaît seulement dans le terme t_1, celui-ci est le seul qui sera modifié par σ. Son mode devient *any*.

   (c) $\mathbf{m_1 = v}$

   On est presque dans la même situation qu'au point précédent, sauf que cette fois-ci, la variable Y peut apparaître dans d'autres termes.

   - Comme les termes $t_2, \ldots, t_p$ sont, soit ground, soit des variables non partagées (et donc différentes de Y), ces termes ne sont pas modifiés par σ.

- Parmi les termes $t_{p+1}, \ldots, t_{p+q}$ par contre, chaque terme qui est une variable partagée, peut être égal à Y et donc, après application de σ, devenir un terme quelconque. Ces termes doivent donc désormais être décrits par le mode $m'_i = a$.

(d) $\underline{\mathbf{m_1 = a}}$

Dans ce cas-ci, tout ce que l'on sait est que le domaine de la substitution σ est compris dans

$$\text{var}(t_1, \ldots, t_p),$$

elle peut associer chaque variable dans t_1 à un sous-terme de $f(t_2, \ldots, t_p)$ et chaque variable dans $t_2, \ldots, t_p$ à un sous-terme de t_1.

Ainsi :

- Chaque terme $t_2, \ldots, t_p$ qui est une variable (qu'elle soit partagée ou pas) peut être unifié avec un sous-terme de t_1 et doit donc être décrit par le mode *any*. Ceci correspond à

$$\forall i \in \{2, \ldots, p\} : m'_i = (v/f \rightarrow a)(m_i).$$

- Chaque terme parmi les $t_{p+1}, \ldots, t_{p+q}$, qui est une variable partagée, peut être contenue dans le domaine de σ et doit, après l'application de σ, être décrite par le mode a. Ceci correspond à

$$\forall i \in \{p+1, \ldots, p+q\} : m'_i = (v \rightarrow a)(m_i).$$

3. $\underline{\exists \mathbf{i} : (\mathbf{i} \in \{2, \ldots, p\} \wedge \mathbf{m_i} \in \{\mathbf{v}, \mathbf{a}\})}$

L'argumentation suivie est analogue à celles que nous avons données. Nous nous limitons à considérer le troisième sous-cas qui est un peu particulier :

(a) $\underline{\mathbf{m_1 = v}}$

Dans ce cas t_1 est une variable, disons Y.
Comme mgu de t_1 et $f(t_2, \ldots, t_p)$, on peut prendre la substitution

$$\sigma = \{Y/f(t_2, \ldots, t_p)\}.$$

On a alors que :

$$\theta\sigma = \{X_1/f(t_2, \ldots, t_p), X_2/t_2\sigma, \ldots, X_p/t_p\sigma, X_{p+1}/t_{p+1}\sigma, \ldots, X_{p+q}/t_{p+q}\sigma\}.$$

Maintenant on peut remarquer que, pour que PROLOG puisse unifier les termes Y et $f(t_2, \ldots, t_p)$, il faut que l'occur-check réussisse, donc que Y n'apparaisse pas dans les termes $t_2, \ldots, t_p$.
Par conséquent :

$$\theta\sigma = \{X_1/f(t_2, \ldots, t_p), X_2/t_2, \ldots, X_p/t_p, X_{p+1}/t_{p+1}\sigma, \ldots, X_{p+q}/t_{p+q}\sigma\}$$

et l'on voit que, en effet, cette substitution peut être décrite par les modes

$$m'_1 = a, m'_2 = m_2, \ldots, m'_p = m_p, \forall i \in \{p+1, \ldots, p+q\} : m'_i = (v \rightarrow a)(m_i).$$

25.3.3 La sémantique abstraite d'un appel de procédure

Considérons

- L'ensemble

$$\text{dtrans} = \{\text{v\_to\_a}, \text{v\_to\_vg}, \text{void}\}$$

- La fonction

$$\text{trans\_v} : \mathcal{M} \times \text{dtrans} \rightarrow \mathcal{P}(\mathcal{M})$$

définie par :

$$\text{trans\_v}(\text{m}, \text{t}) = \text{m} \qquad \text{si } m \neq v$$

$$\text{trans\_v}(v, t) = \{m' \mid m' = \text{if } t = \text{v\_to\_a} \text{ then } a \text{ else if } t = \text{v\_to\_vg} \text{ then } m' \in \{v, g\} \text{ else } m' = v\}$$

- Pour chaque $k \in \mathbb{N}_0$, la fonction
  $\text{ttrans}_k : \mathcal{M}^k \times \mathcal{M}^k \to \text{dtrans}$
  définie par :

$$\text{ttrans}_k(m_1,\ldots,m_k,m'_1,\ldots,m'_k) = \text{if } \exists i : m_i \in \{v,a\} \wedge m'_i = a \text{ then v\_to\_a else if } \exists i : m_i \in \{v,a\} \wedge m'_i = g \text{ then v\_to\_vg else void}$$

La sémantique abstraite d'un appel de procédure peut alors être définie comme suit :

$$\mathcal{F}^* \; [\![p(X_{i_1},\ldots,X_{i_k})]\!] \; \text{env}^* \; \{X_1/m_1,\ldots,X_n/m_n\} \ni \{X_1/m'_1,\ldots,X_n/m'_n\}$$

$$\Leftrightarrow$$

$$\{X_1/m'_{i_1},\ldots,X_k/m'_{i_k}\} \in (\text{env}^* \; p) \; \{X_1/m_{i_1},\ldots,X_k/m_{i_k}\}$$

$$\forall j \notin \{i_1,\ldots,i_k\} : m'_j \in \text{trans\_v}(m_j,T)$$

$$\text{où : } T = \text{ttrans}_k(m_{i_1},\ldots,m_{i_k},m'_{i_1},\ldots,m'_{i_k})$$

Sûreté

Pour démontrer qu'il s'agît ici d'une abstraction sûre, on peut (par symétrie) se limiter au cas d'un appel :

$p(X_1,\ldots,X_k)$.

On doit alors démontrer l'assertion suivante :

Soit $\theta^* = \{X_1/m_1,\ldots,X_n/m_n\} \in \mathcal{S}ubst^*$ une substitution abstraite,
$\theta = \{X_1/t_1,\ldots,X_n/t_n\} \in \mathcal{C}c(\theta^*)$
Soit env* un environnement abstrait
env $\in \mathcal{C}c(\text{env}^*)$.

Soit
$\{X_1/t'_1,\ldots,X_k/t'_k\} \in (\text{env } p)(\{X_1/t_1,\ldots,X_k/t_k\})$
Soit σ la substitution la plus générale telle que :

$$\begin{cases} t'_1 = t_1\,\sigma \\ \vdots \\ t'_k = t_k\,\sigma \end{cases}$$

Alors il existe un tuple $(m'_1,\ldots,m'_n)$ et un $T \in \text{dtrans}$, tel que :

$\{X_1/m'_1,\ldots,X_k/m'_k\} \in (\text{env}^* \; p)\{X_1/m_1,\ldots,X_k/m_k\}$
$T = \text{ttrans}_k(m_1,\ldots,m_k,m'_1,\ldots,m'_k)$
$\forall i \in \{k+1,\ldots,n\} : m'_i = \text{trans\_v}(m_i,T)$
et
$\{X_1/t_1\sigma,\ldots,X_n/t_n\sigma\} \in \mathcal{C}c(\{X_1/m'_1,\ldots,X_n/m'_n\})$.

Démonstration

Comme env $\in \mathcal{C}c(\text{env}^*)$,
$\{X_1/t'_1,\ldots,X_k/t'_k\} \in (\text{env } p)(\{X_1/t_1,\ldots,X_k/t_k\})$
et $\{X_1/t_1,\ldots,X_k/t_k\} \in \mathcal{C}c(\{X_1/m_1,\ldots,X_k/m_k\})$,
on peut trouver $m_1,\ldots,m'_k \in \mathcal{M}$ tels que :
$\{X_1/t'_1,\ldots,X_k/t'_k\} \in \mathcal{C}c(\{X_1/m'_1,\ldots,X_k/m'_k\})$.
Soit alors
$T = \text{ttrans}_k(m_1,\ldots,m_k,m'_1,\ldots,m'_k)$

et pour chaque $i \in \{k+1, \ldots, n\}$:

$m'_i = \text{trans\_v}(m_i, T)$.

On veut maintenant démontrer que

$\{X_1/t_1\ \sigma, \ldots, X_n/t_n\ \sigma\} \in \mathcal{C}c(\{X_1/m'_1, \ldots, X_n/m'_n\})$.

Donc que chaque composant

$X_i/t_i\ \sigma$

est bien abstrait par le mode m'_i.

Sans perte de généralité, il suffit de démontrer ceci pour les deux composants $X_1/t_1\ \sigma$ et $X_n/t_n\ \sigma$.

1. **le composant $X_1/t_1\ \sigma$**

   On sait déjà que

   $\{X_1/t_1\ \sigma, \ldots, X_k/t_k\ \sigma\} \in \mathcal{C}c(\{X_1/m'_1, \ldots, X_k/m'_k\})$.

   Donc, si $m'_1 \in \{g, a, v\}$, le terme $t_1\ \sigma$ est respectivement un terme clos, un terme quelconque ou une variable et peut être décrit par le même mode dans la substitution étendue

   $\{X_1/t_1\ \sigma, \ldots, X_n/t_n\ \sigma\}$.

   Le seul problème qui se pose est quand $m_1 = f$.
   Dans ce cas $t_1\ \sigma$ est une variable non partagée dans la substitution

   $\{X_1/t_1\ \sigma, \ldots, X_k/t_k\ \sigma\}$,

   mais ceci ne signifie pas d'office qu'elle sera toujours non-partagée dans la substitution étendue.

   Comme pour que $t_1\ \sigma$ puisse être une variable, il faut que t_1 soit également une variable, deux cas se présentent :

   (a) **$\mathbf{m_1 = f}$**

   Dans ce cas t_1 est une variable Y qui n'apparaît pas dans les autres termes

   $t_2, \ldots, t_n$.

   Et t'_1 est une variable Z qui n'apparaît pas parmi les

   $t'_2, \ldots, t'_k$.

   Ces variables n'apparaissent donc qu'une seule fois dans la substitution σ, sous la forme

   $\sigma = \{X/Z, \ldots\}$.

   On voit facilement que ceci implique que la variable Z ne peut apparaître dans aucun des termes

   $t_2\ \sigma, \ldots, t_n\ \sigma$.

   (b) **$\mathbf{m_1 = v}$**

   Dans ce cas-ci, l'assertion ne vaut en fait pas...
   Si $m_1 = v$ et $m'_1 = f$, le terme $t_1\ \sigma$ doit (dans le contexte de la substitution

   $\{X_1/t_1\ \sigma, \ldots, X_n/t_n\ \sigma\}$)

   être décrit par le mode v, ni plus, ni moins!

   En toute rigeur il aurait fallu compliquer notre définition encore davantage pour prendre en compte ce cas particulier. Rien que pour voir après que la situation ne se présentera jamais au niveau des calculs!

   En effet, comme nous n'avons dans nos définitions pour la sémantique abstraite, jamais permis qu'un mode v soit transformé en un mode f, tous les environnements abstraits générés seront tel que si $m_1 = v$, m'_1 sera soit g, v ou a.

   Bien que ce soit un sacrilège du point de vue rigeur, il ne nous a pas semblé opportun de compliquer la définition (et par conséquent son implémentation) encore davantage avec la gestion de ce cas dont on sait d'avance que l'on ne s'en servira pas...

2. **le composant $X_n/t_n\ \sigma$**

Considérons pour cette partie de la démonstration, les quatre cas qui peuvent se présenter pour la valeur de m_n.

(a) $\mathbf{m_n = g}$

Dans ce cas t_n est ground et $t_n\ \sigma$ le sera également. Le mode $m'_n = g$ convient donc parfaitement.

(b) $\mathbf{m_n = a}$

Dans ce cas $m'_n = a$, convient à la description de n'importe quel terme, y compris $t_n\ \sigma$.

(c) $\mathbf{m_n = f}$

Dans ce cas, t_n est une variable qui n'apparaît pas dans les termes

$$t_1, \ldots, t_{n-1}.$$

La variable t_n ne peut donc pas apparaître dans le domaine de σ (compris dans $vars(t_1, \ldots, t_k)$), de sorte que

$$t_n\ \sigma = t_n.$$

Comme la variable t_n n'apparaît pas non plus dans le codomaine de σ, cette substitution ne peut pas introduire la variable t_n dans les termes

$$t_1\ \sigma, \ldots, t_{n-1}\ \sigma.$$

Le mode $m'_n = f$ convient donc à la description de $t_n\ \sigma$ dans la substitution

$$\{X_1/t_1\ \sigma, \ldots, X_n/t_n\ \sigma\}.$$

(d) $\mathbf{m_n = v}$

Ce cas est le plus compliqué et demande qu'on considère les trois valeurs que peut prendre

$$T = \text{ttrans}_k(m_1, \ldots, m_k, m'_1, \ldots, m'_k).$$

i. $\mathbf{T}$ = v_to_a

Dans ce cas, le mode $m'_n = a$ convient parfaitement.

ii. $\mathbf{T}$ = v_to_vg

Dans ce cas, m'_n peut prendre les valeurs v et g.
On doit donc démontrer que $t_n\ \sigma$ est effectivement soit une variable, soit ground.

La démonstration est vite faite :

Comme t_n est une variable, il faudrait pour que $t_n\ \sigma$ soit autre que ground ou var, que la sustitution σ applique une variable à quelque chose qui n'est ni ground, ni var.

Pourtant puisque $T \neq$ v_to_a, on sait que :

$$\forall i \in \{1, \ldots, k\} : m_i \in \{v, a\} \Rightarrow m'_i \in \{g, f, v\}.$$

D'où l'on voit que la substitution σ va appliquer chaque variable dans son domaine qui peut être égale à t_n, à un terme ground ou var.

iii. $\mathbf{T}$ = void

Dans ce cas chaque variable partagée dans les termes $t_1, \ldots, t_k$ est appliquée à une variable par la substitution σ.

$t_n\ \sigma$ sera par conséquent toujours une variable, ce qui correspond à $m'_n = v$.

25.3.4 La sémantique abstraite d'un goal

Comme pour la sémantique concrète, la sémantique abstraite d'un goal est définie à l'aide de la sémantique des atomes qui le composent :

$$\mathcal{F}^* \; [\![\alpha_1, \ldots, \alpha_n]\!] \; \text{env}^* \; \theta^* = \bigcup_{\theta' \in \mathcal{F}^*[\![\alpha_1]\!] \; \text{env}^* \; \theta^*} \mathcal{F}^* \; [\![\alpha_2, \ldots, \alpha_n]\!] \; \text{env}^* \; \theta'$$

La sûreté de cette définition est une conséquence directe de la sûreté de la définition de la sémantique abstraite pour les atomes.

25.3.5 La sémantique abstraite d'une procédure

De la même façon, la sémantique abstraite d'une procédure est définie à l'aide de la sémantique des goals qui la composent :

- $\mathcal{F}^* \; [\![p(X_1, \ldots, X_k) : -g]\!] \; \text{env}^* \; \{X_1/m_1, \ldots, X_k/m_k\}$
  $= \mathcal{F}^* \; [\![g]\!] \; \text{env}^* \; \{X_1/m_1, \ldots, X_k/m_k, X_{k+1}/f, \ldots, X_{k+l}/f\}$
  où $X_1, \ldots, X_k, X_{k+1}, \ldots, X_{k+l}$ sont toutes les variables qui apparaissent dans g.
- $\mathcal{F}^* \; [\![p(X_1, \ldots, X_k) : -g_1 \mid \ldots \mid g_n]\!] \; \text{env}^* \; \theta^*$
  $= \bigcup_{i=1}^{n} \mathcal{F}^* \; [\![p(X_1, \ldots, X_k) : -g_i]\!] \; \text{env}^* \; \theta^*$

A nouveau la propriété de sûreté est évidente à vérifier.

25.4 La sémantique abstraite d'un programme

Muni de l'opérateur $\mathcal{F}^*$ qui décrit la sémantique des sous-formules, on peut maintenant définir un opérateur

$$\Xi^* : \mathcal{E}nv^* \rightarrow \mathcal{E}nv^*$$

qui prend une approximation de la sémantique de P et en renvoie une nouvelle comme suit :

$$(\Xi^* \; \text{env}^*) \; p_i \; \theta^* = \mathcal{F} \; [\![proc]\!] \; \text{env}^* \; \theta^*$$

où *proc* est la procédure qui correspond au prédicat p_i.

La sémantique abstraite de P est alors défini comme étant le plus petit point fixe de cet opérateur.

Du fait que notre dérivation de la sémantique est sûre, on peut démontrer que la sémantique concrète sera effectivement une concrétisation de la sémantique abstraite, ce qui est bien une propriété désirée.

26 La traduction de la sémantique abstraite en β-langage

Maintenant que la sémantique abstraite d'un programme PROLOG est définie comme le plus petit point fixe d'un opérateur, on peut procéder à son calcul.

Pour cela, il suffit de reprendre l'algorithme générique du calcul de point fixe qu'on a vu au chapitre 4 et de l'implémenter pour ce problème particulier.

Nous proposons pourtant une approche bien plus simple qui consiste en l'expresssion de la définition de la sémantique abstraite sous forme d'un β-programme.

Tout ce qui est le calcul du point fixe sera alors de façon transparante, être prise en charge par notre interpréteur.

Dans ce chapitre-ci, nous allons spécifier comment cette traduction d'un programme PROLOG en le programme β qui sert à son analyse, peut être effectuée.

26.1 La traduction des fonctions auxilliaires

Nous rappelons que les opérations abstraites ont été définies à l'aide de quelques fonctions auxiliaires (notamment : $unif\_var_n$, $unif\_func_{p,q}$, $trans\_v$ et $ttrans_k$).

Dans un premier temps, il s'agît de traduire chacune de ces fonctions en prédicats du β—langage.

26.1.1 La fonction trans_v

La fonction $trans\_v$ peut être traduite en β-langage comme suit :

```
domain dtrans  {v_to_a, v_to_vg, void};

trans_v(M,M',T:dtrans) :=
     M <> v & M' = M |
     M = v & T = void & M' = v |
     M = v & T = v_to_a & M' = a |
     M = v & T = v_to_vg & M' in {v,g};
```

26.1.2 Les fonctions $ttrans_k$

La famille de fonctions $ttrans_k$ peut être définie à l'aide d'une famille de prédicats $ttrans\_k$ qui s'appellent de façon récursive :

Pour le cas de base ($k = 1$) on peut prendre :

```
ttrans_1(M,M',T:dtrans) :=
     M in {g,f} & T = void |
     M in {v,a} & M' in {f,v} & T = void |
     M in {v,a} & M' = g & T = v_to_vg |
     M in {v,a} & M' = a & T = v_to_a;
```

Et si $k > 1$:

```
ttrans_k(M1,...,Mk,M1',...,Mk',T:dtrans) :=
     ttrans_1(M1,M1',T) & T in {v_to_vg,v_to_a} |
     ttrans_1(M1,M1',void) &  ttrans_k-1(M2,...,Mk,M2',...,Mk',T);
```

26.1.3 Les fonctions unif_var$_n$

Les fonctions *unif_var*$_n$ peuvent être définies à l'aide d'une famille de prédicats

unif_var_n $\subseteq \mathcal{M}^n \times \mathcal{M}^n$

et d'un prédicat auxilliaire

unif_var_aux $\subset \mathcal{M}^3$.

L'idée est que le prédicat auxilliaire détermine les nouvelles valeurs pour m_1' et m_2', après quoi on considère la transition

$(m_1, m_2) \rightarrow (m_1', m_2')$

pour déterminer l'opération à effectuer sur les autres modes m_i.

```
unif_var_aux(M1,M2,M) :=
     M1 = g & M = g |
     M2 = g & M = g |
     M1 <> g & M2 <> g & M1 = a & M = a |
     M1 <> g & M2 <> g & M2 = a & M = a |
     M1 in {f,v} & M2 in {f,v} & M = v;

unif_var_n(M1,M2,...,Mn,M1',M2',...,Mn') :=
     unif_var_aux(M1,M2,M) &
     M1' = M & M2' = M & ttrans_2(M1,M2,M1',M2',T) &
     trans_v(M3,M3',T) & ... & trans_v(Mn,Mn',T);
```

26.1.4 Les fonctions unif_func$_{p,q}$

Pour l'implémentation des fonctions *unif_func*$_{p,q}$ on s'est inspiré sur les trois tableaux à la page (146).

Deux familles de prédicats *testg*$_p$ et *testva*$_p$ permettent de déterminer lequel de ces tableaux s'applique.

Ensuite la deuxième, troisième et quatrième colonne dans chacun de ces tableaux est calculée.

```
testg_p(M1,...,Mp) :=
    M1 = g & ... & Mp = g;

testva_p(M1,...,Mp) :=
    M1 in {v,a} | ... | Mp in {v,a};

trans_vf_to_a(M,M') :=
     M in {v,f} & M' = a |
     M in {g,a} & M' = M;

domain dtrans_col3  {all_to_g3, vf_to_a3, void3};

col3(M,T:dtrans_col3) :=
     M = g & T = all_to_g3 |
     M in {f,v} & T = void3 |
     M = a & T = vf_to_a3;

trans_col3(M,M',T:dtrans_col3) :=
     T = all_to_g3 & M' = g |
     T = vf_to_a3 & M in {v,f} & M' = a |
     T = vf_to_a3 & M in {g,a} & M' = M |
     T = void3 & M' = M;

domain dtrans_col4  {v_to_a4, v_to_gv4, vf_to_a4, void4};
```

```
col4_cas2(M,T:dtrans_col4) :=
     M in {g,f} & T = void4 |
     M in {v,a} & T = v_to_a4;

col4_cas3(M,T:dtrans_col4) :=
     M = g & T = v_to_gv4 |
     M = f & T = void4 |
     M = f & T = v_to_a4 |
     M = a & T = vf_to_a4;

trans_col4(M,M',T:dtrans_col4) :=
     T = v_to_a4 & M = v & M' = a |
     T = v_to_a4 & M <> v & M' = M |
     T = v_to_gv4 & M = v & M' in {g,v} |
     T = v_to_gv4 & M <> v & M' = M |
     T = vf_to_a4 & M in {v,f} & M' = a |
     T = vf_to_a4 & M in {g,a} & M' = M |
     T = void4 & M' = M ;


unif_func_p_q(M1,...,Mp+q,M1',...,Mp+q') :=
     testg_p-1(M2) &
         M1'=g & ... & Mp'=g &
         ttrans_1(M1,M1',T) &
         trans_v(Mp+1,Mp+1',T) & ... & trans_v(Mp+q,Mp+q',T) |
     not testg_p-1(M2) & not testva_p-1(M2) &
         col3(M1,T3) & col4_cas2(M1,T4) &
         trans_col3(M2,M2',T3) & ... & trans_col3(Mp,Mp',T3) &
         trans_col4(Mp+1,Mp+1',T4) & ... & trans_col4(Mp+q,Mp+q',T4) &
         trans_vf_to_a(M1,M1') |
     testva_p-1(M2) &
         col3(M1,T3) & col4_cas3(M1,T4) &
         trans_col3(M2,M2',T3) & ... & trans_col3(Mp,Mp',T3) &
         trans_col4(Mp+1,Mp+1',T4) & ... & trans_col4(Mp+q,Mp+q',T4) &
         trans_vf_to_a(M1,M1');
```

26.2 Les règles de traduction

Maintenant que l'on sait comment traduire les fonctions auxiliaires, il est chose facile de traduire un programme PROLOG en le programme β qui servira à son analyse.

Cette traduction est réalisée à l'aide des quelques fonctions spécifiées ci-après :

26.2.1 La traduction d'une unification de variables

trad_uvar : string → int → int → int → string
s = trad_uvar(nom_atome,n,i,j)

paramètres

nom_atome	:	le nom de l'atome à traduire : ce nom est égal à p_i_j, si l'atome en question est le *j-ième* atome de la *i-ième* clause du prédicat *p*
n	:	l'arité du prédicat contenant l'atome
i , j	:	les entiers tels que l'atome à traduire est $X_i = X_j$.

définition

s =

```
nom_atome(m1,..,mn,m1',..,mn') :=
        unif_var_n(mi,mj,m1,..,mi-1,mi+1,..,mj-1,mj+1,..,mn,
                   mi',mj',m1',..,mi-1',mi+1',..,mj-1',mj+1',..,mn');
```

26.2.2 La traduction d'une unification fonctionnelle

trad_ufunc : string → int → (int list) → string
s = trad_ufunc(nom_atome,n,[$i_1, \ldots, i_p$])

paramètres

nom_atome	:	le nom de l'atome à traduire
n	:	l'arité du prédicat contenant l'atome
$[i_1, \ldots, i_p]$	:	l'atome en question a la forme $X_{i_1} = f(X_{i_2}, \ldots, X_{i_p})$

définition

$\{j_1, ..., j_q\} = \{1, \ldots, n\} - \{i_1, \ldots, i_p\}$

s =

```
nom_atome(m1,..,mn,m1',..,mn') :=
        unif_func_p_q(mi1,..,mip,mj1,..,mjq,
                      mi1',..,mip',mj1',..,mjq');
```

26.2.3 La traduction d'un appel de prédicat

trad_appel : string → int → string → (int list) → string
s = trad_appel(nom_atome,n,p,[$i_1, \ldots, i_k$])

paramètres

nom_atome	:	le nom de l'atome à traduire
n	:	l'arité du prédicat contenant l'atome
p	:	le nom du prédicat appelé
$[i_1, \ldots, i_k]$	:	l'atome en question est $p(X_{i_1}, \ldots, X_{i_k})$

définition

$\{j_1, ..., j_l\} = \{1, \ldots, n\} - \{i_1, \ldots, i_k\}$

s =

```
nom_atome(m1,..,mn,m1',..,mn') :=
        p(mi1,..,mik,mi1',..,mik') &
        ttrans_k(mi1,..,mik,mi1',..,mik',T) &
        trans_v(mj1,mj1',T) &
        ... &
        trans_v(mjl,mjl',T);
```

26.2.4 La traduction d'un atome

trad_atome : string → int → atome → string
s = trad_atome(nom_atome,n,a)

paramètres

nom_atome	:	le nom de l'atome à traduire
a	:	l'atome en question

définition

s =

match a with

$X_i = X_j \rightarrow$ trad_uvar(nom_atome,n,i,j)

$X_{i_1} = f(X_{i_2}, \ldots, X_{i_p}) \rightarrow$ trad_ufunc(nom_atome,n,$[i_1, \ldots, i_p]$)

$p(X_{i_1}, \ldots, X_{i_p}) \rightarrow$ trad_appel(nom_atome,n,$[i_1, \ldots, i_p]$)

26.2.5 La traduction d'un goal

trad_goal : string → int → goal → string
s = trad_goal(nom_goal,k,$[a_1, \ldots, a_q]$)

paramètres

k	:	l'arité du prédicat pour lequel on considère le goal
nom_clause	:	le nom du goal à traduire, ce nom est p_i s'il s'agît du i-ième goal pour le prédicat p.
$[a_1, \ldots, a_q]$	:	les atomes du goal

définition

n = le numéro de variable maximal dans $a_1, \ldots, a_q$

s_0 =

```
nom_goal(m1,..,mk,m1(q),..,mn(q)) :=
        mk+1=f&..&mn=f,
        nom_goal_1(m1,..,mn,m1',..,mn') &
        ... &
        nom_goal_q(m1(q-1),..,mn(q-1),m1(q),..,mn(q)).
```

s_i = trad_atome(nom_goal_i,n, a_i)

$s = s_0 + s_1 + \ldots + s_q$

26.2.6 La traduction d'une procédure

trad_proc : string → int → goal list → string
s = trad_proc(nom_pred,n,$[g_1, \ldots, g_k]$)

paramètres

nom_pred	:	le nom du prédicat à traduire
n	:	l'arité du prédicat
$[g_1, \ldots, g_k]$	:	les goals de la procédure

définition

s_0 =

```
nom_pred(m1,..,mn,m1',..,mn') :=
        nom_pred_1(m1,..,mn,m1',..,mn') |
        ... |
        nom_pred_k(m1,..,mn,m1',..,mn').
```

s_i = trad_goal(nom_pred_i,n, g_i)

$s = s_0 + s_1 + \ldots + s_k$

26.2.7 La traduction d'un programme

trad_prog : prog → string
s = **trad_prog**($[pr_1, \ldots, pr_k]$)

paramètres

s	:	la traduction du programme source
$[pr_1, \ldots, pr_k]$	:	les procédures du programme source

définition

$pr_i = (p_i, n_i, \text{goals}_i)$

s_i = trad_proc(p_i, n_i, goals_i)

$s' = s_1 + \ldots + s_k$

s est la chaîne s' à laquelle on a ajouté toutes les définitions de domaine et de prédicats auxilliaires qu'on a utilisées dans la chaîne s.

27 Un exemple

Afin d'automatiser la traduction d'un programme β, nous avons écrit un petit programme CAML dont on peut trouver le code source en annexe.

Avant qu'on ne puisse utiliser ce programme, il faut que le programme PROLOG source soit d'abord représenté en termes des types CaML suivants :

- Un programme est une suite de procédures :
  prog = proc list
- Une procédure est caractérisée par un nom, une arité et une suite de goals :
  proc = string * int * (goal list)
- Un goal est une suite d'atomes :
  goal = atome list
- Un atome est soit une unification de variables, une unification fonctionnelle ou un appel de procédure :

```
atome = U_VAR    of int * int
      | U_FUNC   of int * string * (int list)
      | APPEL    of string * (int list)
```

Ainsi, le petit programme PROLOG ci-dessous qui implémente l'addition et la mutiplication pour les nombres naturels :

$$\begin{cases} \text{natPlus}(X1, X2, X3) : -X1 = 0, X2 = X3. \\ \text{natPlus}(X1, X2, X3) : -X1 = s(X4), X3 = s(X5), \text{natPlus}(X4, X2, X5). \\ \\ \text{natMul}(X1, X2, X3) : -X1 = 0, X3 = 0. \\ \text{natMul}(X1, X2, X3) : -X1 = s(X4), \text{natMul}(X4, X2, X5), \text{natPlus}(X2, X5, X3). \end{cases}$$

peut être représenté sous la forme :

```
let prog_test =
   [("natPlus",3,
             [
             [UFUNC (1,"O",[]); UVAR (2,3)];
             [UFUNC (1,"s",[4]); UFUNC(3,"s",[5]);
                                 APPEL("natPlus",[4;2;5])]
             ]);
    ("natMul",3,
             [
             [UFUNC(1,"O",[]); UFUNC(3,"O",[])];
             [UFUNC(1,"s",[4]); APPEL("natMul",[4;2;5]);
                                APPEL("natPlus",[2;5;3])]
             ])
   ];;
```

Pour générer le programme β, il suffit alors de faire un simple appel :

print_string (trad_prog prog_test);;

27.1 Le programme β

```
domain {g,f,v,a};

domain dtrans  {v_to_a, v_to_vg, void};

domain dtrans_col3  {all_to_g3, vf_to_a3, void3};

domain dtrans_col4  {v_to_a4, v_to_gv4, vf_to_a4, void4};
```

```
natPlus(M1,M2,M3,M1',M2',M3') :=
    natPlus_1(M1,M2,M3,M1',M2',M3') |
    natPlus_2(M1,M2,M3,M1',M2',M3');

natPlus_1(M1,M2,M3,M1'',M2'',M3'') :=
    natPlus_1_1(M1,M2,M3,M1',M2',M3') &
    natPlus_1_2(M1',M2',M3',M1'',M2'',M3'');
natPlus_1_1(M1,M2,M3,M1',M2',M3') :=
    unif_func_1_2(M1,M2,M3,M1',M2',M3');
natPlus_1_2(M1,M2,M3,M1',M2',M3') :=
    unif_var_3(M2,M3,M1,M2',M3',M1');

natPlus_2(M1,M2,M3,M1''',M2''',M3''') := M4=f & M5=f &
    natPlus_2_1(M1,M2,M3,M4,M5,M1',M2',M3',M4',M5') &
    natPlus_2_2(M1',M2',M3',M4',M5',M1'',M2'',M3'',M4'',M5'') &
    natPlus_2_3(M1'',M2'',M3'',M4'',M5'',M1''',M2''',M3''',M4''',M5''');
natPlus_2_1(M1,M2,M3,M4,M5,M1',M2',M3',M4',M5') :=
    unif_func_2_3(M1,M4,M2,M3,M5,M1',M4',M2',M3',M5');
natPlus_2_2(M1,M2,M3,M4,M5,M1',M2',M3',M4',M5') :=
    unif_func_2_3(M3,M5,M1,M2,M4,M3',M5',M1',M2',M4');
natPlus_2_3(M1,M2,M3,M4,M5,M1',M2',M3',M4',M5') :=
    natPlus(M4,M2,M5,M4',M2',M5') &
    ttrans_3(M4,M2,M5,M4',M2',M5',T) &
    trans_v(M1,M1',T) & trans_v(M3,M3',T);


natMul(M1,M2,M3,M1',M2',M3') :=
    natMul_1(M1,M2,M3,M1',M2',M3') |
    natMul_2(M1,M2,M3,M1',M2',M3');

natMul_1(M1,M2,M3,M1'',M2'',M3'') :=
    natMul_1_1(M1,M2,M3,M1',M2',M3') &
    natMul_1_2(M1',M2',M3',M1'',M2'',M3'');
natMul_1_1(M1,M2,M3,M1',M2',M3') :=
    unif_func_1_2(M1,M2,M3,M1',M2',M3');
natMul_1_2(M1,M2,M3,M1',M2',M3') :=
    unif_func_1_2(M3,M1,M2,M3',M1',M2');

natMul_2(M1,M2,M3,M1''',M2''',M3''') := M4=f & M5=f &
    natMul_2_1(M1,M2,M3,M4,M5,M1',M2',M3',M4',M5') &
    natMul_2_2(M1',M2',M3',M4',M5',M1'',M2'',M3'',M4'',M5'') &
    natMul_2_3(M1'',M2'',M3'',M4'',M5'',M1''',M2''',M3''',M4''',M5''');
natMul_2_1(M1,M2,M3,M4,M5,M1',M2',M3',M4',M5') :=
    unif_func_2_3(M1,M4,M2,M3,M5,M1',M4',M2',M3',M5');
natMul_2_2(M1,M2,M3,M4,M5,M1',M2',M3',M4',M5') :=
    natMul(M4,M2,M5,M4',M2',M5') &
    ttrans_3(M4,M2,M5,M4',M2',M5',T) &
    trans_v(M1,M1',T) & trans_v(M3,M3',T);
natMul_2_3(M1,M2,M3,M4,M5,M1',M2',M3',M4',M5') :=
    natPlus(M2,M5,M3,M2',M5',M3') &
    ttrans_3(M2,M5,M3,M2',M5',M3',T) &
    trans_v(M1,M1',T) & trans_v(M4,M4',T);
```

```
//predicats auxilliares :

trans_v(M,M',T:dtrans) :=
     M <> v & M' = M |
     M = v & T = void & M' = v |
     M = v & T = v_to_a & M' = a |
     M = v & T = v_to_vg & M' in {v,g};

trans_vf_to_a(M,M') :=
     M in {v,f} & M' = a |
     M in {g,a} & M' = M;

unif_var_aux(M1,M2,M) :=
     M1 = g & M = g |
     M2 = g & M = g |
     M1 <> g & M2 <> g & M1 = a & M = a |
     M1 <> g & M2 <> g & M2 = a & M = a |
     M1 in {f,v} & M2 in {f,v} & M = v;

col3(M,T:dtrans_col3) :=
     M = g & T = all_to_g3 |
     M in {f,v} & T = void3 |
     M = a & T = vf_to_a3;

trans_col3(M,M',T:dtrans_col3) :=
     T = all_to_g3 & M' = g |
     T = vf_to_a3 & M in {v,f} & M' = a |
     T = vf_to_a3 & M in {g,a} & M' = M |
     T = void3 & M' = M;

col4_cas2(M,T:dtrans_col4) :=
     M in {g,f} & T = void4 |
     M in {v,a} & T = v_to_a4;

col4_cas3(M,T:dtrans_col4) :=
     M = g & T = v_to_gv4 |
     M = f & T = void4 |
     M = f & T = v_to_a4 |
     M = a & T = vf_to_a4;

trans_col4(M,M',T:dtrans_col4) :=
     T = v_to_a4 & M = v & M' = a |
     T = v_to_a4 & M <> v & M' = M |
     T = v_to_gv4 & M = v & M' in {g,v} |
     T = v_to_gv4 & M <> v & M' = M |
     T = vf_to_a4 & M in {v,f} & M' = a |
     T = vf_to_a4 & M in {g,a} & M' = M |
     T = void4 & M' = M ;

unif_var_3(M1,M2,M3,M1',M2',M3') :=
     unif_var_aux(M1,M2,M) &
     M1' = M & M2' = M & ttrans_2(M1,M2,M1',M2',T) &
     trans_v(M3,M3',T);

unif_func_2_3(M1,M2,M3,M4,M5,M1',M2',M3',M4',M5') :=
     testg_1(M2) &
```

```
        M1'=g & M2'=g &
        ttrans_1(M1,M1',T) &
        trans_v(M3,M3',T) & trans_v(M4,M4',T) & trans_v(M5,M5',T) |
    not testg_1(M2) & not testva_1(M2) &
        col3(M1,T3) & col4_cas2(M1,T4) &
        trans_col3(M2,M2',T3) &
        trans_col4(M3,M3',T4) & trans_col4(M4,M4',T4) &
        trans_col4(M5,M5',T4) &
        trans_vf_to_a(M1,M1') |
    testva_1(M2) &
        col3(M1,T3) & col4_cas3(M1,T4) &
        trans_col3(M2,M2',T3) &
        trans_col4(M3,M3',T4) & trans_col4(M4,M4',T4) &
        trans_col4(M5,M5',T4) &
        trans_vf_to_a(M1,M1');

unif_func_1_2(M1,M2,M3,M1',M2',M3') :=
        M1'=g &
        ttrans_1(M1,M1',T) &
        trans_v(M2,M2',T) & trans_v(M3,M3',T) ;

testg_1(M1) :=
    M1 = g;

testva_1(M1) :=
    M1 in {v,a};

ttrans_3(M1,M2,M3,M1',M2',M3',T:dtrans) :=
    ttrans_1(M1,M1',T) & T in {v_to_vg,v_to_a} |
    ttrans_1(M1,M1',void) & ttrans_2(M2,M3,M2',M3',T);

ttrans_2(M1,M2,M1',M2',T:dtrans) :=
    ttrans_1(M1,M1',T) & T in {v_to_vg,v_to_a} |
    ttrans_1(M1,M1',void) & ttrans_1(M2,M2',T);

ttrans_1(M,M',T:dtrans) :=
    M in {g,f} & T = void |
    M in {v,a} & M' in {f,v} & T = void |
    M in {v,a} & M' = g & T = v_to_vg |
    M in {v,a} & M' = a & T = v_to_a;
```

27.2 L'analyse de *natPlus*

La réponse à la requête

? natPlus

contient pas moins de 129 lignes.

On nous pardonnera sans doute si nous nous limiterons dans cette discussion à considérer le cas restreint où $M_1 = g$ et $M_2 \in \{g, f\}$.

La partie pertinente de la table est alors :

```
(M1:g, M2:g, M3:g, M1':g, M2':g, M3':g)
(M1:g, M2:g, M3:f, M1':g, M2':g, M3':g)
(M1:g, M2:g, M3:f, M1':g, M2':g, M3':a)
(M1:g, M2:g, M3:v, M1':g, M2':g, M3':g)
```

```
(M1:g, M2:g, M3:v, M1':g, M2':g, M3':a)
(M1:g, M2:g, M3:a, M1':g, M2':g, M3':g)
(M1:g, M2:g, M3:a, M1':g, M2':g, M3':a)
(M1:g, M2:f, M3:g, M1':g, M2':g, M3':g)
(M1:g, M2:f, M3:f, M1':g, M2':v, M3':v)
(M1:g, M2:f, M3:f, M1':g, M2':v, M3':a)
(M1:g, M2:f, M3:v, M1':g, M2':v, M3':v)
(M1:g, M2:f, M3:v, M1':g, M2':v, M3':a)
(M1:g, M2:f, M3:a, M1':g, M2':a, M3':a)
```

On remarquera qu'on y trouve pas mal de ligne superflues. En effet, si par la troisième ligne, on apprend que une substition d'entrée (g, g, f) peut être transformée en une substitution (g, g, a), la deuxième ligne qui nous indique la possibilité d'une substitution de sortie (g, g, g) ne nous apprend strictement plus rien. D'une certaine manière, on peut dire que la deuxième ligne est subsummée par la troisième.

Nous avons essayé d'enlever ces ligne subsummées dans le β-programme même, mais cela nous a conduit au paradoxe dont nous avons parlé au chapitre 8.

Si l'on vire les lignes superflues à la main, on trouve :

```
(M1:g, M2:g, M3:g, M1':g, M2':g, M3':g)
(M1:g, M2:g, M3:f, M1':g, M2':g, M3':a)
(M1:g, M2:g, M3:v, M1':g, M2':g, M3':a)
(M1:g, M2:g, M3:a, M1':g, M2':g, M3':a)

(M1:g, M2:f, M3:g, M1':g, M2':g, M3':g)
(M1:g, M2:f, M3:f, M1':g, M2':v, M3':a)
(M1:g, M2:f, M3:v, M1':g, M2':v, M3':a)
(M1:g, M2:f, M3:a, M1':g, M2':a, M3':a)
```

- La première ligne
  $$(g, g, g) \rightarrow (g, g, g),$$
  ne viendra sans doute pas comme une grosse surprise.
- La deuxième ligne
  $$(g, g, f) \rightarrow (g, g, a),$$
  est déjà plus intéressante, puisqu'elle ne donne pas la transformation
  $$(g, g, f) \rightarrow (g, g, g)$$
  attendue.

  Si on refait l'analyse à la main, on comprend vite l'origine de cette imprécision :

  Pour l'analyse de la clause non récursive on trouve :

$$\begin{array}{l}\{X_1/g, X_2/g, X_3/f\} \\ \qquad \text{natPlus}(X_1, X_2, X_3) :- \\ \{X_1/g, X_2/g, X_3/f\} \\ \qquad\qquad X_1 = 0, \\ \{X_1/g, X_2/g, X_3/f\} \\ \qquad\qquad X_2 = X_3 \\ \{X_1/g, X_2/g, X_3/g\} \\ \qquad\qquad . \\ \{X_1/g, X_2/g, X_3/g\}\end{array}$$

C'est-à-dire, la transformation $(g, g, f) \rightarrow (g, g, g)$.

Pour ce qui est de l'analyse de la clause récursive, on obtient :

$$\{X_1/g, X_2/g, X_3/f\}$$
$$\text{natPlus}(X_1, X_2, X_3) :-$$
$$\{X_1/g, X_2/g, X_3/f, X_4/f, X_5/f\}$$
$$X_1 = s(X_4),$$
$$\{X_1/g, X_2/g, X_3/f, X_4/g, X_5/f\}$$
$$X_3 = s(X_5),$$
$$\{X_1/g, X_2/g, X_3/a, X_4/g, X_5/f\}$$
$$\text{natPlus}(X_4, X_2, X_5)$$

Arrivé à ce point, on sait par ce que l'on a obtenu pour la clause non récursive que la $\{X_4/g, X_2/g, X_5/f\}$ en entrée peut se transformer en une $\{X_4/g, X_2/g, X_5/g\}$. Le mode pour X_5 devient donc g, mais (et c'est là l'origine du problème) notre substitution abstraite n'a gardé aucune trace du fait que $X_3 = s(X_5)$. On ne sait donc, avec le domaine abstrait considéré, pas déduire que X_3 devient ground aussi et on doit se contenter de :

$$\{X_1/g, X_2/g, X_3/a, X_4/g, X_5/g\}$$

$$\{X_1/g, X_2/g, X_3/a\}$$

- Les deux lignes suivantes :

  $(g, g, v) \rightarrow (g, g, a)$ et $(g, g, a) \rightarrow (g, g, a)$

  sont analogues au cas précédent.
- La transformation $(g, f, g) \rightarrow (g, g, g)$ nous donne le résultat attendu.
- Les deux lignes suivantes :

  $(g, f, f) \rightarrow (g, v, a)$ et $(g, f, v) \rightarrow (g, v, a)$

  donnent également un résultat aussi précis que possible.
  Pour s'en assurer, il suffit de considérer la transformation

  $\{X_1/s(0), X_2/Y_1, X_3/Y_2\} \rightarrow \{X_1/s(0), X_2/Y_1, X_3/s(Y_1)\}$.
- Il en va de même pour la dernière ligne :

  $(g, f, a) \rightarrow (g, a, a)$.

  Là il suffit de considérer la transformation :

  $\{X_1/0, X_2/Y_1, X_3/s(Y_2)\} \rightarrow \{X_1/0, X_2/s(Y_2), X_3/s(Y_2)\}$.

Conclusions

Ceci termine la présentation de notre logiciel.
Nous croyons avoir mené à bien la tâche qui nous était proposée, c'est à dire d'implémenter la β-sémantique (cfr. [4]) pour les domaines finis en optimisant le calcul du point fixe avec l'algorithme qu'on retrouve dans [1].

Nous avons démontré que ce point de départ est effectivement implémentable de façon efficace.

Plus précisément,

- nous avons écrit la syntaxe pour notre langage et un parseur qui sait réagir de façon intelligente à des erreurs syntaxiques
- nous avons développé une représentation des relations qui permet à la fois une gestion efficace de la mémoire et une implémentation rapide des opérations relationnelles
- nous avons appliqué l'algorithme générique du calcul de point fixe à notre problème particulier.

En plus, nous avons démontré l'utilité de notre interpréteur pour l'interprétation abstraite de programmes PROLOG.

Le résultat de tout ce travail est plus que satisfaisant : même un problème aussi lourd que celui des huit reines peut être résolu en un temps tout à fait raisonnable et avec une consommation de mémoire qui est minimale grâce aux mécanismes du partage des nœuds et l'enlèvement des nœuds non référencés que nous avons implémentés.

Pourtant il reste encore beaucoup à faire.

Dans notre travail, on peut trouver plusieurs idées quant à l'optimisation ou l'extension du logiciel. Nous en récapitulons les plus importantes :

- Au niveau de la représentation des relations d'autres pistes restent à explorer :
  Sans doute serait-il intéressant de combiner les parties positives et négatives d'une birelation en un seul arbre dont les nœuds terminaux seront alors *TRUE*, *FALSE* et *INCONNU*.
  Pour des domaines de plus d'éléments, d'autres représentations plus économe en mémoire doivent être développées.
- Une solution efficace pour l'adaptation dynamique de la taille maximale de la table des nœuds doit être trouvée.
- Au niveau des opérations relationnelles, le temps de réponse pourrait bénéficier de l'implémentation d'une mémoire cache dans laquelle les résultats des quelques dernières opérations relationnelles seraient stockés.
- Un meilleur algorithme pour le numérotage des variables, permettrait à la fois de réaliser des économies mémoire (en choisissant les numéros des variables d'une telle manière qu'un partage de nœuds maximal devient possible) et une réduction du temps de réponse (en minimisant le nombre de non-monotonies dans les appels de littéraux) (cfr. chapitre 16)

- Au niveau du calcul de point fixe, il pourrait être intéressant d'implémenter un graphe des dépendances étendu (cfr. section 18.4) de sorte que l'algorithme puisse voir sur ce graphe si une conjonction ou un littéral doit être réévalué sans qu'il doive pour cela descendre au niveau des prédicats.
- Une autre suggestion au niveau du calcul, est d'implémenter la version *orientée requête* de la sémantique décrite au paragraphe (18.5). Ceci permettrait à l'interpréteur de n'évaluer que les parties des prédicats dont il a effectivement besoin au lieu de les calculer entièrement. Le gain en temps de réponse et en consommation de mémoire qui peut en résulter pourrait être considérable.
- Au niveau de l'interface il serait peut-être bon d'en avoir une...

  Même pour une version non-interactive comme celle-ci, on devrait permettre à l'utilisateur de faire des requêtes sur la partie positive ou négative d'une sous-formule générale (et pas seulement sur un prédicat comme c'est le cas dans la version actuelle).
- Comme nous avons remarqué au chapitre (8), le β-langage donne parfois des résultats très surprenants. L'étendue et les implications de ceci restent encore à explorer.
- Et bien sûr la question la plus fondamentale reste ouverte : A quel point ce que nous avons fait dans ce travail peut-il être généralisé pour des domaines infinis ?

Nous espérons que ceux qui vont continuer notre travail, trouverons l'architecture que nous avons conçue assez modulaire et bien documentée pour pouvoir implémenter quelques unes de ces extensions.

Références

[1] Baudouin Le Charlier and Pascal Van Hentenryck. *A General Fixpoint Algorithm for Abstract Interpretation.*

[2] Marc-Michel Corsini and Antoine Rauzy. *First Experiment with Toupie.*

[3] Marc-Michel Corsini and Antoine Rauzy. *Toupie User's Manual.*

[4] Naji Habra and Baudouin Le Charlier. *Semantic framework for Composite Logic Programming*, 1995.

[5] J. .W Lloyd. *Foundation of Logic Programming.* Springer-Verlag, 1987. Second, Extended Edition.

[6] Leon Sterling and Ehud Shapiro with a foreword by David H. D. Warren. *The Art of Prolog (Advanced Programming Techniques).* The MIT Press, Cambridge, Massachussetts, London, England. Second Edition.

Cinquième partie

Annexes

A Le code

A.1 makefile

```
OBJS=  y.tab.o  lex.yy.o principal.o tliste.o \
       numerotage.o typage.o trelation.o tables.o \
       printrelation.o  pointfixe.o  operations.o

WFLAGS=  -Wall -W -Wshadow -Waggregate-return  -Winline

DCFLAGS=$(WFLAGS) -g
OCFLAGS=$(WFLAGS) -DNDEBUG -O3

CFLAGS=$(OCFLAGS)

CC= gcc $(CFLAGS) -c

beta: $(OBJS)
        gcc $(CFLAGS) -o ../beta $(OBJS) -lm
        cp ../beta .

a.out: $(OBJS)
        gcc $(CFLAGS)  $(OBJS)

operations.o : operations.c
        $(CC) operations.c

pointfixe.o : pointfixe.c
        $(CC) pointfixe.c

printrelation.o : printrelation.c
        $(CC) printrelation.c

tables.o : tables.c
        $(CC) tables.c

trelation.o : trelation.c
        $(CC) trelation.c

numerotage.o : numerotage.c
        $(CC) numerotage.c

typage.o : typage.c
        $(CC) typage.c

principal.o: principal.c
        $(CC) principal.c

y.tab.o: y.tab.c
        gcc -g -c y.tab.c

lex.yy.o: lex.yy.c  y.tab.h
        gcc -g -c lex.yy.c

y.tab.c y.tab.h: parse.y
        byacc -d   parse.y

lex.yy.c: scan.l
        flex -8 scan.l

tliste.o: tliste.c tliste.h
        $(CC) tliste.c

clean:
        rm -f core '#'*  *~ *.log *.aux *.dvi *.toc *.o lex.yy.c y.tab.* \
        y.output a.out  2>/dev/null

all:
        make clean; make
```

A.2 numerotage.c

Ce module exporte deux fonctions pour affecter des numéros respectivement aux prédicats et aux variables du programme source.

```
#include <malloc.h>
#include <stdio.h>

#include "tables.h"
```

Codage

```
static void NumeroterPredicatsDeConjonction(TConjonction *Conj){
  TLiteral* Lit;
  TAppel* Appel;
  int Index, NParAbstr, NParEff, i;

  for (i=0;i<Conj->Literaux->Count;i++){
    Lit = (TLiteral*) GetItem(Conj->Literaux,i);
    Appel = Lit->Appel;
    Index = GetNrPredicat(Appel->NomPred);
    if (Index<0){
      printf("erreur : ligne %d : le predicat %s n'est pas declare",
             Appel->NLigne,Appel->NomPred);
      exit(1);
    }

    NParAbstr = GetPredicat(Index)->ParamTypes->Count;
    NParEff = Appel->ParamsEffs->Count;

    if (NParAbstr != NParEff){
      printf("erreur : ligne %d : l'arite du predicat %s est %d",
             Appel->NLigne, Appel->NomPred, NParAbstr);
      exit(1);
    }
    Appel->Pred = Index;
  }
}

void NumeroterPredicats(void){
  TPredicat* Pred;
  TConjonction* Conj;
  int i,j;

  for (i=0;i<PredTable->Count;i++){
    Pred = (TPredicat*)GetItem(PredTable,i);
    for (j=0;j<Pred->Conjonctions->Count;j++){
      Conj=(TConjonction*)GetItem(Pred->Conjonctions,j);
      NumeroterPredicatsDeConjonction(Conj);
    }
  }
}

static TListe* NomsVarsConjonction(TConjonction* Conj){ /*SET[String]*/
  TListe* Result;
  TLiteral* Literal;
  TAppel* Appel;
  TListe* ParamEffs;
  TContrainte* Contr;
  TComparVar* Compar;
  TAppartenance* Appart;
  TVarOuConst* ParEff;
  int i,j;

  Result = CreateListe();
```

```
  for (i=0;i<Conj->Literaux->Count;i++){
    Literal=(TLiteral*)GetItem(Conj->Literaux,i);
    Appel = Literal->Appel;
    ParamEffs = Appel->ParamsEffs;
    for (j=0;j<ParamEffs->Count;j++){
      ParEff=(TVarOuConst*)GetItem(ParamEffs,j);
      if (ParEff->T==Var)
        AddCopyToStringSet(Result,ParEff->Def.Var->NomVar);
    }
  }


  for (i=0;i<Conj->Contraintes->Count;i++){
    Contr=(TContrainte*)GetItem(Conj->Contraintes,i);
    if (Contr->T == ComparVar){
      Compar = Contr->Def.Comparaison;
      AddCopyToStringSet(Result,Compar->NomVar1);
      AddCopyToStringSet(Result,Compar->NomVar2);
    }
    else{
      Appart = Contr->Def.Appartenance;
      AddCopyToStringSet(Result,Appart->NomVar);
    }
  }


  return Result;
}


static void NumeroterVarsConj(TConjonction* Conj,
                      TListe* NomVarPred, TListe* VarPred,
                      TListe* NomVarConj, TListe* VarConj){
  TListe *NomVars,*Vars;
  TLiteral *Literal;
  TAppel *Appel;
  TVarOuConst *ParEff;
  int Nr;
  TContrainte *Contr;
  TComparVar *Compar;
  TAppartenance *Appart;
  int i,j;


  NomVars = AppendStringListe(NomVarPred,NomVarConj);
  Vars = AppendIntListe(VarPred,VarConj);


  for (i=0;i<Conj->Literaux->Count;i++){
    Literal = (TLiteral*) GetItem(Conj->Literaux,i);
    Appel = Literal->Appel;


    for (j=0;j<Appel->ParamsEffs->Count;j++){
      ParEff=(TVarOuConst*)GetItem(Appel->ParamsEffs,j);
      if (ParEff->T == Var){
        Nr =  GetIndexString(NomVars,ParEff->Def.Var->NomVar);
        ParEff->Def.Var->Var = *(int*) GetItem(Vars,Nr);
      }
    }


  }
```

```
  for (i=0;i<Conj->Contraintes->Count;i++){
    Contr=(TContrainte*)GetItem(Conj->Contraintes,i);
    if (Contr->T == ComparVar){
      Compar = Contr->Def.Comparaison;
      Compar->Var1 = GetInt(Vars,GetIndexString(NomVars,Compar->NomVar1));
      Compar->Var2 = GetInt(Vars,GetIndexString(NomVars,Compar->NomVar2));
    }
    else{
      Appart = Contr->Def.Appartenance;
      Appart->Var = GetInt(Vars,GetIndexString(NomVars,Appart->NomVar));
    }
  }
  SortIntListe(VarConj);
  Conj->VarsQuants = CopyIntListe(VarConj);
  DestroyListe(NomVars);
  DestroyListe(Vars);
}
static void NumeroterVarsPred(int NrPred){
  TPredicat *Pred;
  TListe *NomVarPred,*NomVarConjTout,*NomVarConj; /* SEQ[String] */
  TParamType *ParTyp;
  TConjonction *Conj;
  int i,j;
  TListe *VarConj; /* SEQ[Integer] */
  Pred = GetPredicat(NrPred);
  NomVarPred = CreateListe();
  for (i=0; i<Pred->ParamTypes->Count; i++){
    ParTyp=(TParamType*)GetItem(Pred->ParamTypes,i);
    AddCopyToStringListe(NomVarPred,ParTyp->NomVar);
    NVariables++;
    AddCopyToStringListe(NomVarTable,ParTyp->NomVar);
    AddToIntListe(Pred->Vars,NVariables-1);
  }
  Pred->VarsSorted = CopyIntListe(Pred->Vars);
  SortIntListe(Pred->VarsSorted);
  for (i=0;i<Pred->Conjonctions->Count;i++){
    Conj=(TConjonction*)GetItem(Pred->Conjonctions,i);
    NomVarConjTout = NomsVarsConjonction(Conj);
    NomVarConj = DiffStringList(NomVarConjTout,NomVarPred);
    DestroyListe(NomVarConjTout);
    VarConj = CreateListe();
    for (j=0;j<NomVarConj->Count;j++){
      NVariables++;
      AddCopyToStringListe(NomVarTable,GetString(NomVarConj,j));
      AddToIntListe(VarConj,NVariables-1);
    }
    NumeroterVarsConj(Conj,NomVarPred,Pred->Vars,NomVarConj,VarConj);
    DestroyListe(NomVarConj);
    DestroyListe(VarConj);
  }
  DestroyListe(NomVarPred);
}
void NumeroterVars(void){
  int i;
  for (i=0;i<PredTable->Count; i++)
    NumeroterVarsPred(i);
}
```

A.3 operations.h

```
#ifndef OPERATIONS_H
#define OPERATIONS_H

#include "tliste.h"
#include "trelation.h"

TRelation CreateEgalite(int x,int y);
TRelation CreateInegalite(int x,int y);
TRelation VarEgalConst(int X,int C);
TRelation CreateAppartenance(int v, TListe *C);

TRelation Et(TRelation r1, TRelation r2, int keep1, int keep2);
TRelation Ou(TRelation r1, TRelation r2, int keep1, int keep2);
TRelation Not(TRelation r, int keep);
TRelation Diff(TRelation r1, TRelation r2, int keep1, int keep2);
TRelation IlExiste(TRelation r, TListe* VarListe, int keep);
TRelation PourTout(TRelation r, TListe* VarListe,int keep);
TRelation Projection(TRelation r,TListe *VarListe, int keep);

typedef enum {SVar,SConst} ETypeSubst;

typedef struct {
  ETypeSubst T;
  int Var;
  int Value;
} TSubst;

TRelation Substituer(TRelation R, int n, TSubst* Subst,int keep);

#endif
```

A.4 operations.c

Ce module implémente les opérations relationnelles

```
#include <stdio.h>
#include <malloc.h>
#include <assert.h>
#include "trelation.h"
#include "tliste.h"
#define ILEXISTE 0
#define POURTOUT 1


int GetCardVar(int v);
TRelation IlExiste(TRelation r, TListe* VarListe, int keep);
TRelation PourTout(TRelation r, TListe* VarListe, int keep);
typedef enum {SVar,SConst} ETypeSubst;


typedef struct {
  ETypeSubst T;
  int Var;
  int Value;
} TSubst;
```

Codage

```
TRelation CreateEgalite(int x,int y) {
  int i,j,card;
  TRelation *filsx,*filsy;


  if (x==y) return TRUE;
  if (x>y){
    i=x;
    x=y;
    y=i;
  }
  card = GetCardVar(x);
  assert(card == GetCardVar(y));
  filsx = (TRelation*)malloc(card*sizeof(TRelation));
  for (i=0;i<card;i++){
    filsy = (TRelation*)malloc(card*sizeof(TRelation));
    for (j=0;j<card;j++)
      filsy[j] = FALSE;
    filsy[i] = TRUE;
    filsx[i] = CreateRelation(y,filsy);
  }
  return CreateRelation(x,filsx);
}


TRelation CreateInegalite(int x,int y) {
  int i,j,card;
  TRelation *filsx,*filsy;
```

```
  if (x==y) return FALSE;
  if (x>y){
    i=x;
    x=y;
    y=i;
  }
  card = GetCardVar(x);
  assert(card == GetCardVar(y));
  filsx = (TRelation*)malloc(card*sizeof(TRelation));
  for (i=0;i<card;i++){
    filsy = (TRelation*)malloc(card*sizeof(TRelation));
    for (j=0;j<card;j++)
      filsy[j] = TRUE;
    filsy[i] = FALSE;
    filsx[i] = CreateRelation(y,filsy);
  }
  return CreateRelation(x,filsx);
}

TRelation VarEgalConst(int X,int C){
  TRelation *Fils;
  int i;

  Fils = (TRelation*)malloc(GetCardVar(X)*sizeof(TRelation));
  for(i= 0;i<GetCardVar(X);i++)
    Fils[i]=FALSE;
  Fils[C]=TRUE;
  return CreateRelation(X,Fils);
};

TRelation CreateAppartenance(int v, TListe *C) {
                  /* C=SEQ[Integer] */
  int card, i;
  TRelation *fils;

  card = GetCardVar(v);
  fils = (TRelation*) malloc(card * sizeof(TRelation));
  for (i=0; i<card; i++)
    fils[i] = FALSE;
  for (i=0; i< C->Count; i++)
    fils[GetInt(C,i)] = TRUE;
  return CreateRelation(v,fils);
}

TRelation Et(TRelation r1, TRelation r2,int keep1,int keep2) {
  int x1,x2;
  TRelation *nd1, *nd2, *newnd;
  TRelation out=FALSE; /*pour eviter un avertissement*/
  int i;

  if (r1==FALSE || r2==FALSE){
    if (keep1 == 0) DestroyRelation(r1);
    if (keep2 == 0) DestroyRelation(r2);
    return FALSE;
  }
  if (r1==TRUE){
    if (keep2==0)
      return r2;
    else return CopyRelation(r2);
  }
  if (r2==TRUE){
    if (keep1==0)
      return r1;
    else return CopyRelation(r1);
  }
```

```
  x1=GetLabel(r1);
  x2=GetLabel(r2);



  nd1=GetFils(r1);
  nd2=GetFils(r2);
  if (x1<x2) {
    newnd=(TRelation*)malloc(GetCardVar(x1)*sizeof(TRelation));
    for (i=0 ; i<GetCardVar(x1); i++)
      newnd[i]=Et(nd1[i],r2,1,1);
    out=CreateRelation(x1,newnd);
  }
  else if (x1>x2) {
    newnd=(TRelation*)malloc(GetCardVar(x2)*sizeof(TRelation));
    for (i=0 ; i<GetCardVar(x2); i++)
      newnd[i]=Et(r1,nd2[i],1,1);
    out=CreateRelation(x2,newnd);
  }
  else if (x1==x2) {
    newnd=(TRelation*)malloc(GetCardVar(x1)*sizeof(TRelation));
    for (i=0 ; i<GetCardVar(x1); i++)
      newnd[i]=Et(nd1[i],nd2[i],1,1);
    out=CreateRelation(x1,newnd);
  }
  if (keep1==0) DestroyRelation(r1);
  if (keep2==0) DestroyRelation(r2);
  return out;
}



TRelation Ou(TRelation r1, TRelation r2, int keep1, int keep2){
  int x1,x2;
  TRelation *nd1, *nd2, *newnd;
  TRelation out=FALSE; /*pour eviter un avertissement*/
  int i;



  if (r1==TRUE || r2==TRUE){
    if (keep1 == 0) DestroyRelation(r1);
    if (keep2 == 0) DestroyRelation(r2);
    return TRUE;
  }
  if (r1==FALSE){
    if (keep2==0)
      return r2;
    else return CopyRelation(r2);
  }
  if (r2==FALSE){
    if (keep1==0)
      return r1;
    else return CopyRelation(r1);
  }



  x1=GetLabel(r1);
  x2=GetLabel(r2);
```

```
  nd1=GetFils(r1);
  nd2=GetFils(r2);
  if (x1<x2) {
    newnd=(TRelation*)malloc(GetCardVar(x1)*sizeof(TRelation));
    for (i=0 ; i<GetCardVar(x1); i++)
      newnd[i]=Ou(nd1[i],r2,1,1);
    out=CreateRelation(x1,newnd);
  }
  else if (x1>x2) {
    newnd=(TRelation*)malloc(GetCardVar(x2)*sizeof(TRelation));
    for (i=0 ; i<GetCardVar(x2); i++)
      newnd[i]=Ou(r1,nd2[i],1,1);
    out=CreateRelation(x2,newnd);
  }
  else if (x1==x2) {
    newnd=(TRelation*)malloc(GetCardVar(x1)*sizeof(TRelation));
    for (i=0 ; i<GetCardVar(x1); i++)
      newnd[i]=Ou(nd1[i],nd2[i],1,1);
    out=CreateRelation(x1,newnd);
  }
  if (keep1==0) DestroyRelation(r1);
  if (keep2==0) DestroyRelation(r2);
  return out;
}

TRelation Not(TRelation r,int keep){
  int x;
  TRelation *nd,*newnd;
  int i;

  if (r==TRUE) return FALSE;
  if (r==FALSE) return TRUE;
  x=GetLabel(r);
  nd=GetFils(r);
  newnd=(TRelation*)malloc(GetCardVar(x)*sizeof(TRelation));
  for (i=0 ; i <= GetCardVar(x)-1 ; i++)
    newnd[i]=Not(nd[i],1);
  if (keep==0) DestroyRelation(r);
  return CreateRelation(x,newnd);
}

TRelation Diff(TRelation r1, TRelation r2, int keep1, int keep2){
  TRelation r;

  r=Not(r2,keep2);
  return Et(r1,r,keep1,0);
}


TRelation PourTout_IlExiste_core(int which, TRelation r, TListe* VarListe, int n, int nmax, int keep){
  int x, v, i;
  TRelation *nd, *newnd, res;
  TRelation out=FALSE;

  if (n==nmax)
    if (keep==0)
      return r;
    else
      return CopyRelation(r);

  if (r==TRUE || r==FALSE) return r;
  x=GetLabel(r);
  nd=GetFils(r);
  v=GetInt(VarListe,n);
```

```
  if (v<x)
    out=PourTout_IlExiste_core(which,r,VarListe,n+1,nmax,1);
  else if (v>x) {
    newnd=(TRelation*) malloc(GetCardVar(x)*sizeof(TRelation));
    for (i=0; i<GetCardVar(x); i++)
      switch (which) {
      case ILEXISTE   : newnd[i]=IlExiste(nd[i],VarListe,1); break;
      case POURTOUT   : newnd[i]=PourTout(nd[i],VarListe,1); break;
      }
    out=CreateRelation(x,newnd);
  }
  else if (v==x) {
    res = CopyRelation(nd[0]);
    for (i=0;i<GetCardVar(x);i++)
      switch (which) {
      case ILEXISTE   : res=Ou(res,nd[i],0,1); break;
      case POURTOUT   : res=Et(res,nd[i],0,1); break;
      }
    out=PourTout_IlExiste_core(which,res,VarListe,n+1,nmax,0);
  }

  if (keep==0) DestroyRelation(r);
  return out;
}

TRelation IlExiste(TRelation r, TListe* VarListe, int keep){
  return PourTout_IlExiste_core(ILEXISTE,r,VarListe,0,Count(VarListe),keep);
}

TRelation PourTout(TRelation r, TListe* VarListe, int keep){
  return PourTout_IlExiste_core(POURTOUT,r,VarListe,0,Count(VarListe),keep);
}

static TRelation SubstituerLabels(TRelation R, int n, TSubst* Subst, int keep){
  TRelation *NouveauFils, *Fils;
  int i, found, NouveauLabel, X;
  TRelation out;

  if (R == TRUE || R == FALSE) return R;

  Fils=GetFils(R);
  X=GetLabel(R);

  found = -1;
  for (i=0;i<n;i++)
    if (Subst[i].Var==X){
      found = i;
      break;
    }

  if (found != -1)
    if (Subst[found].T == SConst)
      return SubstituerLabels(Fils[Subst[found].Value],n,Subst,1);
    else
      NouveauLabel = Subst[found].Value;
  else
    NouveauLabel = X;

  NouveauFils = (TRelation*)malloc(GetCardVar(X)*sizeof(TRelation));
  for (i=0;i<GetCardVar(X);i++)
    NouveauFils[i] = SubstituerLabels(Fils[i],n,Subst,1);
  out=CreateRelation(NouveauLabel,NouveauFils);
  if (keep == 0) DestroyRelation(R);
  return out;
}
```

```
static TRelation Reorganiser(TRelation R,int keep){
  int X,i;
  TRelation  Fils, Out;

  if (R==TRUE||R==FALSE) return R;
  Out = FALSE;
  X=GetLabel(R);
  for(i=0;i<GetCardVar(X);i++){
    Fils = Reorganiser(GetFils(R)[i],1);
    Out = Ou(Out,Et(VarEgalConst(X,i),Fils,0,0),0,0);
  }
  if (keep==0) DestroyRelation(R);
  return Out;
}

TRelation Substituer(TRelation R, int n, TSubst* Subst, int keep){
  TRelation Out;

  Out = SubstituerLabels(R,n,Subst,keep);
  return Reorganiser(Out,0);
}
```

A.5 parse.y

Ce module permet de parser un fichier source

```
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "tables.h"
#include <math.h>
#define make(T) (T*)malloc(sizeof(T))
#define DOMAINE      0
#define APPARTENANCE 1

extern TListe* PredAppeles;
extern TConjonction* Request;
Boolean domainedefaut=FALSE, err;
int linecount=1;
int cste_case=APPARTENANCE;
extern char *actual_line;
int binf,bsup;
int i,j,k;

void erreur(int i, char* s){
  printf("[-%d-] Ligne %d : ",i,linecount);
  switch (i) {
  case 3  : {printf("nom de domaine deja utilise (%s).\n",s);} ; break;
  case 4  : {printf("domaine par defaut defini une seconde fois.\n");} ; break;
  case 5  : {printf("nom de predicat deja utilise (%s).\n",s);} ; break;
  case 6  : {printf("nom de constante ou d'intervalle deja utilise (%s).\n",s);} ; break;
  case 7  : {printf("meme nom de variable plusieurs fois dans les parametres (%s).\n",s);} ; break;
  case 8  : {printf("constante non declaree (%s).\n",s);} ; break;
  case 9  : {printf("les constantes du domaines ne sont pas de même type (%s declaree ailleurs).\n",s);} br
  case 10 : {printf("l'intervalle est vide (%s).\n",s);} ; break;
  case 11 : {printf("nom d'intervalle inexistant ou bornes incorrectes (%s[%d..%d]).\n",s,binf,bsup);} brea
  case 12 : {printf("nom d'intervalle deja existant (%s).\n",s);} ; break;
  case 13 : {printf("domaine non declare (%s).\n",s);} ; break;
  case 14 : {printf("domaine par defaut non declare (%s non typable).\n",s);} ; break;
  case 15 : {printf("predicat appele mais non declare (%s).\n",s);} ; break;
  case 18 : {printf("enumeration vide.\n");} ; break;
  default : { printf("erreur"); } ; break;}
  printf("%s\n",actual_line);
  exit(1);
}

char* cst_str(char* nom, int index) {
  char *s;
  int i, n;
  s=(char*) malloc(100*sizeof(char*));
  strcpy(s,nom);
  s[strlen(nom)]='[';
  s[strlen(nom)+(int) log10((double) index)+2]=']';
  s[strlen(nom)+(int) log10((double) index)+3]='\0';
  for (i=(int) log10((double) index) ; i>=0 ; i--) {
    n= (index % ((int) pow((double) 10, (double) (i+1))))
      / ((int) pow((double) 10, (double) (i)));
    s[strlen(nom)+(int) log10((double) index)-i+1]=(char) n+48;
  }
  return s;
}
%}

%union {
  char* string;
  int integer;
```

```
  TDomaine* type_definition_domaine_par_defaut;
  TDomaine* type_definition_domaine_nomme;
  TDomaine* type_description_domaine_nomme;
  TDescrDomaine* type_description_domaine;
  TListe* type_description_domaine_enumere;
  TDomIntervalle* type_description_domaine_intervalle;
  TListe* type_liste_constantes;
  TPredicat* type_definition_predicat;
  TListe* type_liste_param_types;
  TListe* type_liste_param_types_non_vide;
  TParamType* type_param_type;
  TListe* type_disjonction;
  TConjonction* type_conjonction;
  TLiteral* type_literal;
  TAppel* type_appel_predicat;
  TContrainte* type_atome;
  EOperateur type_operateur;
  TListe* type_liste_parametres_effectifs;
  TListe* type_liste_parametres_effectifs_non_vide;
  TVar* type_nom_variable;
  TVarOuConst* type_var_ou_const;
  TConst* type_constante;
  TConstIndexee* type_constante_indexee;
}

%token <string> IDMIN IDMAJ
%token <integer> ENTIER
%token DOMAIN IN POINTPOINT DECLPRED EQU NEQ LT LE GT GE NOT

%type <string> nom_constante
%type <string> nom_intervalle
%type <string> nom_domaine
%type <string> nom_predicat
%type <type_liste_param_types> liste_param_types
%type <type_liste_param_types_non_vide> liste_param_types_non_vide
%type <type_param_type> param_type
%type <type_definition_domaine_par_defaut> definition_domaine_par_defaut
%type <type_definition_domaine_nomme> definition_domaine_nomme
%type <type_description_domaine> description_domaine
%type <type_description_domaine_enumere> description_domaine_enumere;
%type <type_description_domaine_intervalle> description_domaine_intervalle
%type <type_liste_constantes> liste_constantes
%type <type_definition_predicat> definition_predicat
%type <type_disjonction> disjonction
%type <type_conjonction> conjonction
%type <type_literal> literal
%type <type_appel_predicat> appel_predicat
%type <type_atome> atome
%type <type_operateur> operateur
%type <type_liste_parametres_effectifs> liste_parametres_effectifs
%type <type_liste_parametres_effectifs_non_vide> liste_parametres_effectifs_non_vide
%type <type_nom_variable> nom_variable
%type <type_var_ou_const> var_ou_const
%type <type_constante> constante
%type <string> constante_elem
%type <type_constante_indexee> constante_indexee

%%

start : liste_definition
;

liste_definition : definition |  liste_definition definition
;
```

```
definition : definition_domaine | definition_predicat
;

definition_domaine : definition_domaine_par_defaut | definition_domaine_nomme
;

definition_domaine_par_defaut : DOMAIN description_domaine ';' {
  if (Nom_of_Domain_Exist("")<0) {
    $$=make(TDomaine);
    $$->Nom=(char*) malloc(1);
    strcpy($$->Nom,"");
    $$->Definition=$2;
    if ($$->Definition->T==Enumere) {
      $$->Count=Count($$->Definition->Def.DescrEnumere);}
    else {
      $$->Count=$$->Definition->Def.DescrIntervalle->BSup
        - $$->Definition->Def.DescrIntervalle->BInf + 1;}
    Add(DomTable,$$);
  }
  else { $$=NULL;erreur(4,"");}
};

definition_domaine_nomme : DOMAIN nom_domaine description_domaine ';' {
  if (Nom_of_Domain_Exist($2)<0) {
    $$=make(TDomaine);
    $$->Nom=$2;
    $$->Definition=$3;
    if ($$->Definition->T==Enumere) {
      $$->Count=Count($$->Definition->Def.DescrEnumere);}
    else {
      $$->Count=$$->Definition->Def.DescrIntervalle->BSup
        - $$->Definition->Def.DescrIntervalle->BInf + 1;}
    Add(DomTable,$$);
  }
  else {erreur(3,$2);}
};

nom_domaine : IDMIN  {$$=$1;}
            | IDMAJ  {$$=$1;};

description_domaine : description_domaine_enumere {
                        $$=make(TDescrDomaine);
                        $$->T=Enumere;
                        $$->Def.DescrEnumere=$1;}
                    | description_domaine_intervalle {
                        $$=make(TDescrDomaine);
                        $$->T=Intervalle;
                        $$->Def.DescrIntervalle=$1;}
;

description_domaine_enumere : '{' liste_constantes '}' {
  j=Nom_of_Constante_Exist_OutThere(GetItem($2,0));
  for (i=1 ; i<$2->Count ; i++) {
    k=Nom_of_Constante_Exist_OutThere(GetItem($2,i));
    if (k!=j)
      if (cste_case==DOMAINE) erreur(6,GetItem($2,i));
      else erreur(9,GetItem($2,i));
  }
  $$=$2;
};
```

```
description_domaine_intervalle : nom_intervalle '[' ENTIER POINTPOINT ENTIER ']' {
  if (cste_case==DOMAINE) if (Nom_of_Intervalle_Exist($1)>=0) erreur(12,$1);
  if ($5<$3) {erreur(10,$1);}
  else {
    $$=make(TDomIntervalle);
    $$->Nom=$1;
    $$->BInf=$3;
    $$->BSup=$5;
  }
};


nom_intervalle : IDMIN  {$$=$1;}
               | IDMAJ  {$$=$1;};


liste_constantes : nom_constante
                     {$$=CreateListe();
                      Add($$,$1);}
                 | liste_constantes ',' nom_constante {
                     if ((GetIndexString($1,$3)==-1)  || (Nom_of_Constante_Exist_OutThere($3)==-1)) {
                       Add($1,$3);
                       $$=$1;}
                     else erreur(6,$3);
                 };


definition_predicat : nom_predicat '(' liste_param_types ')' DECLPRED disjonction ';' {
  if (GetNrPredicat($1)==-1) {
    $$=make(TPredicat);
    $$->Nom=$1;
    $$->ParamTypes=$3;
    $$->Conjonctions=$6;
    $$->Vars=CreateListe();
    Add(PredTable,$$);
  }
  else {erreur(5,$1);}
};


nom_predicat : IDMIN
                 {$$=$1;}
             | IDMAJ
                 {$$=$1;};


liste_param_types : /* epsillon */
                      {$$=CreateListe();}
                  | liste_param_types_non_vide
                      {$$=$1;};


liste_param_types_non_vide : param_type {
                               $$=CreateListe();
                               Add($$,$1);}
                           | liste_param_types_non_vide ',' param_type {
                             if (Nom_of_Param_Exist($3->NomVar,$1)<0) {
                               Add($1,$3);
                               $$=$1;}
                             else {erreur(7,$3->NomVar);}
                           };
```

```
param_type :  nom_variable {
                if (Nom_of_Domain_Exist("")>=0) {
                  $$=make(TParamType);
                  $$->NomVar=$1->NomVar;
                  $$->NomDomaine=(char*) malloc(1);
                  $$->Domaine=Numero_de_Domaine("");
                  strcpy($$->NomDomaine,"");}
                else {erreur(14,$1->NomVar);}
                }
           | nom_variable ':' nom_domaine {
             if (Nom_of_Domain_Exist($3)>=0) {
               $$=make(TParamType);
               $$->NomVar=$1->NomVar;
               $$->NomDomaine=$3;
               $$->Domaine=Numero_de_Domaine($3);}
             else {erreur(13,$3);}
           }
;

disjonction :  conjonction {
                 $$=CreateListe();
                 Add($$,$1);}
             | disjonction '|' conjonction {
                 Add($1,$3);
                 $$=$1;};

conjonction : literal {
                 $$=make(TConjonction);
                 $$->Contraintes=CreateListe();
                 $$->Literaux=CreateListe();
                 $$->VarsQuants=CreateListe();
                 Add($$->Literaux,$1);
               }
            | atome {
                 $$=make(TConjonction);
                 $$->Contraintes=CreateListe();
                 $$->Literaux=CreateListe();
                 $$->VarsQuants=CreateListe();
                 Add($$->Contraintes,$1);
            }
            | conjonction '&' literal {
              Add($1->Literaux,$3);
              $$=$1;}
            | conjonction '&' atome {
              Add($1->Contraintes,$3);
              $$=$1;}
;
```

```
literal : appel_predicat {
            $$=make(TLiteral);
            $$->Signe=POS;
            $$->Appel=$1;
            $$->FirstTime=1;
            $$->VarsLit=CreateListe();
            $$->VarsDef=CreateListe();
            $$->RelContr=FALSE; }
| NOT appel_predicat {
            $$=make(TLiteral);
            $$->Signe=NEG   ;
            $$->Appel=$2;
            $$->FirstTime=1;
            $$->VarsLit=CreateListe();
            $$->VarsDef=CreateListe();
            $$->RelContr=FALSE; }
;
```

```
appel_predicat : nom_predicat '(' liste_parametres_effectifs ')' {
  $$=make(TAppel);
  $$->NomPred=$1;
  $$->NLigne=linecount;
  $$->ParamsEffs=$3;}
;
```

```
atome : nom_variable IN description_domaine {
          int temp, i, *j;
          TDomaine* D;
```

```
  if ($3->T==Intervalle) {
    temp=Nom_of_Intervalle_Exist($3->Def.DescrIntervalle->Nom,
                                 $3->Def.DescrIntervalle->BInf,
                                 $3->Def.DescrIntervalle->BSup);
    if (temp>-1) {
      $$=make(TContrainte);
      $$->T=Appartenance;
      $$->Def.Appartenance=make(TAppartenance);
      $$->Def.Appartenance->NomVar=$1->NomVar;
      $$->Def.Appartenance->Domaine=temp;
      $$->Def.Appartenance->NLigne=linecount;
      $$->Def.Appartenance->Constantes=CreateListe();
      D=GetItem(DomTable,temp);
      for (i=$3->Def.DescrIntervalle->BInf;
           i<=$3->Def.DescrIntervalle->BSup;
           i++ ) {
        j=make(int);
        *j=i-D->Definition->Def.DescrIntervalle->BInf;
        Add($$->Def.Appartenance->Constantes,j);
      }
    }
    else {
      binf=$3->Def.DescrIntervalle->BInf;
      bsup=$3->Def.DescrIntervalle->BSup;
      erreur(11,$3->Def.DescrIntervalle->Nom);
    }
  }
  else { /* Cas enumere */
    $$=make(TContrainte);
    $$->T=Appartenance;
    $$->Def.Appartenance=make(TAppartenance);
    $$->Def.Appartenance->NomVar=$1->NomVar;
    $$->Def.Appartenance->Constantes=CreateListe();
    temp=Get_Domaine_of_Constante_Nommee((char*) GetItem($3->Def.DescrEnumere,0));
    $$->Def.Appartenance->Domaine=temp;
    $$->Def.Appartenance->NLigne=linecount;
    D=(TDomaine*) GetItem(DomTable,temp);
    Reset($3->Def.DescrEnumere);
    while (!EOL($3->Def.DescrEnumere)) {
      j=make(int);
      *j=GetIndexString(D->Definition->Def.DescrEnumere,
                        (char*) GetNext($3->Def.DescrEnumere));
      if (*j==-1) {erreur(8,Extract($3->Def.DescrEnumere,$3->Def.DescrEnumere->Current));}
      else {
        Add($$->Def.Appartenance->Constantes,j);
      }
    }
  }
 }
| nom_variable operateur var_ou_const {
  TDomaine* D;
  char* c;
  int *j, i, k, temp;
```

```
$$=make(TContrainte);
if ($3->T==Var) {
  $$->T=ComparVar;
  $$->Def.Comparaison=make(TComparVar);
  $$->Def.Comparaison->NomVar1=$1->NomVar;
  $$->Def.Comparaison->Op=$2;
  $$->Def.Comparaison->NomVar2=$3->Def.Var->NomVar;
  $$->Def.Comparaison->NLigne=linecount;
}
else {
  $$->T=Appartenance;
  $$->Def.Appartenance=make(TAppartenance);
  $$->Def.Appartenance->NomVar=$1->NomVar;
  $$->Def.Appartenance->Constantes=CreateListe();
  if ($3->Def.Const->T==Elementaire) {
    if ($2==EQ) {
      $$->Def.Appartenance->Domaine=$3->Def.Const->Dom;
      $$->Def.Appartenance->NLigne=linecount;
      D=(TDomaine*) GetItem(DomTable,$3->Def.Const->Dom);
      j=make(int);
      *j=$3->Def.Const->Const;
      Add($$->Def.Appartenance->Constantes,j);
    }
    else {
      $$->Def.Appartenance->Domaine=$3->Def.Const->Dom;
      $$->Def.Appartenance->NLigne=linecount;
      D=(TDomaine*) GetItem(DomTable,$3->Def.Const->Dom);
      k=$3->Def.Const->Const;
      for (i=0 ; i<Count(D->Definition->Def.DescrEnumere) ; i++) {
        if (i!=k) {
          j=make(int);
          *j=i;
          Add($$->Def.Appartenance->Constantes,j);
        }
      }
    }
  }
  else {
    temp=$3->Def.Const->Dom;
    if (temp>-1) {
      $$->Def.Appartenance->Domaine=temp;
      $$->Def.Appartenance->NLigne=linecount;
      D=(TDomaine*) GetItem(DomTable,temp);
      if ($2==EQ) {
        j=make(int);
        *j=$3->Def.Const->Const;
        Add($$->Def.Appartenance->Constantes,j);
      }
      else {
        for (i=D->Definition->Def.DescrIntervalle->BInf;
             i<=D->Definition->Def.DescrIntervalle->BSup;
             i++ ) {
          if (i!=$3->Def.Const->Def.ConstIndexee->Index) {
            j=make(int);
            *j=i-D->Definition->Def.DescrIntervalle->BInf;
            Add($$->Def.Appartenance->Constantes,j);
          }
        }
      }
    }
    else erreur(8,$3->Def.Const->Def.ConstIndexee->Nom);
  }
}
```

```
      }
;

operateur : EQU {$$=EQ;}
          | NEQ {$$=NE;} ;

liste_parametres_effectifs : /* epsillon */
             {$$=CreateListe();
             }
         | liste_parametres_effectifs_non_vide
             {$$=$1;
             };

liste_parametres_effectifs_non_vide : var_ou_const
              {$$=CreateListe();
              Add($$,$1);}
            | liste_parametres_effectifs_non_vide ',' var_ou_const
              {Add($1,$3);
              $$=$1;
              };

nom_variable : IDMAJ  {
              $$=make(TVar);
              $$->NomVar=$1;
              };

nom_constante : IDMIN  {$$=$1;};

var_ou_const : nom_variable
                 {$$=make(TVarOuConst);
                  $$->T=Var;
                  $$->Def.Var=$1;}
             | constante {
                  if (Declared_Constante($1)==1) {
                    $$=make(TVarOuConst);
                    $$->T=Const;
                    $$->Def.Const=$1;}
                  else {
                    if ($1->T==Elementaire){
                      erreur(8,$1->Def.ConstElem);$$=NULL;}
                    if ($1->T==Indexee) {
                      erreur(8,"Prout!");
                    }
                  }
             }
;

constante : constante_elem {
              TDomaine *Dom;
              TListe *StringList;

              $$=make(TConst);
              $$->T=Elementaire;
              $$->Def.ConstElem=$1;
              $$->Dom=Get_Domaine_of_Constante_Nommee($1);
              if ($$->Dom == -1) erreur(8,$1);
              Dom=GetDomaine($$->Dom);
              StringList=Dom->Definition->Def.DescrEnumere;
              $$->Const=GetIndexString(StringList,$1);
}
          | constante_indexee {
              TDomaine *Dom;
              TListe *StringList;
```

```
                $$=make(TConst);
                $$->T=Indexee;
                $$->Def.ConstIndexee=$1;
                $$->Dom=Nom_of_Intervalle_Exist($1->Nom,$1->Index,$1->Index);
                if ($$->Dom==-1) erreur(8,cst_str($1->Nom,$1->Index));
                Dom=GetDomaine($$->Dom);
                $$->Const=$1->Index - Dom->Definition->Def.DescrIntervalle->BInf ;
}
;

constante_elem : nom_constante {
                $$=$1;}
;

constante_indexee : nom_intervalle '[' ENTIER ']' {
    $$=make(TConstIndexee);
    $$->Nom=$1;
    $$->Index=$3;}
;

%%

char* yytext;

int yyerror(char* s)
{
  printf("erreur on ligne %d : %s before %s\n",linecount,s,yytext);
  exit(1);
  return 1;
}
```

A.6 pointfixe.c

Ce module exporte une seule fonction qui permet de calculer la réponse à une requête.

```
#include <malloc.h>
#define make(T) (T*)malloc(sizeof(T))
#include <stdio.h>
#include "tables.h"
#include "operations.h"

int nrPlus, nrPlus1, nrPlus2;
TPredicat *predPlus1, *predPlus2 , *predPlus;
```

Codage

```
typedef struct{
  ESigne Signe;
  TPredicat* Predicat;
  TListe *Utilise, *UtilisePar;
} TSommet;

static TListe* Dg;

static TSommet *SommetAppelant;

static void Calculer(ESigne Signe, TPredicat* Pred);

static TRelation ContraintesToRel(TListe *L) {
  TContrainte* Contr;
  TRelation Out, R;
  TComparVar *Comp;
  TAppartenance *Appart;
  int i;

  Out = TRUE;

  for (i=0;i<L->Count;i++){
    Contr = (TContrainte*) GetItem(L,i);
    if (Contr->T==ComparVar) {
      Comp = Contr->Def.Comparaison;
      if (Comp->Op==EQ)
        R=CreateEgalite(Comp->Var1,Comp->Var2);
      else
        R=CreateInegalite(Comp->Var1,Comp->Var2);
    }
    else {
      Appart = Contr->Def.Appartenance;
      R=CreateAppartenance(Appart->Var,Appart->Constantes);
    }
    Out = Et(Out,R,0,0);
  }
  return Out;
}

static TRelation EvalLiteral(ESigne Signe, TLiteral* L){
  TAppel *Appel;
  TPredicat *P;
  TListe *ParEff,*VarsPred;
  TRelation Out;
  TSubst *Subst;
  int i,n;
  TVarOuConst *Par;

  Appel = L->Appel;
  P = GetPredicat(Appel->Pred);
  ParEff = Appel->ParamsEffs;
  VarsPred = P->Vars;
```

```
  n=VarsPred->Count;
  Subst = (TSubst*) malloc(n * sizeof(TSubst));
  for (i=0;i<n;i++){
    Subst[i].Var=GetInt(VarsPred,i);
    Par = (TVarOuConst*)GetItem(ParEff,i);
    if (Par->T == Var){
      Subst[i].T = SVar;
      Subst[i].Value = Par->Def.Var->Var;
    }
    else{
      Subst[i].T = SConst;
      Subst[i].Value = Par->Def.Const->Const;
    }
  }

  if (Signe == L->Signe){
    Calculer(POS,P);
    Out = P->Pos;
  }
  else{
    Calculer(NEG,P);
    Out = P->Neg;
  }

  Out = Substituer(Out,n,Subst,1);
  free(Subst);
  return Out;
}

static TRelation EvalConjonction(ESigne Signe, TConjonction* C){
  TLiteral* L;
  TRelation ContrRel;
  TRelation Out;
  int i;

  ContrRel = ContraintesToRel(C->Contraintes);
  if (Signe == POS){
    Out = ContrRel;
    for (i=0;i<C->Literaux->Count;i++){
      L=(TLiteral*)GetItem(C->Literaux,i);
      Out = Et(Out,EvalLiteral(POS,L),0,0);
    }
    Out = IlExiste(Out,C->VarsQuants,0);
  }
  else{
    Out = Not(ContrRel,0);
    for (i=0;i<C->Literaux->Count;i++){
      L=(TLiteral*)GetItem(C->Literaux,i);
      Out = Ou(Out,EvalLiteral(NEG,L),0,0);
    }
    Out = PourTout(Out,C->VarsQuants,0);
  }
  return Out;
}

static TRelation EvalPredicat(ESigne Signe, TPredicat* P){
  TConjonction *C;
  TRelation Out;
  int i;
```

```
  if (Signe == POS) {
    Out = FALSE;
    for (i=0;i<P->Conjonctions->Count;i++){
      C=(TConjonction*)GetItem(P->Conjonctions,i);
      Out = Ou(Out,EvalConjonction(POS,C),0,0);
    };
  }
  else{
    Out = TRUE;
    for (i=0;i<P->Conjonctions->Count;i++){
      C=(TConjonction*)GetItem(P->Conjonctions,i);
      Out = Et(Out,EvalConjonction(NEG,C),0,0);
    }
  }
  return Out;
}

static void Remove(TSommet* Sommet){
  int i,j;
  TSommet *S;

  for(j=0;j<Sommet->Utilise->Count;j++){
    S=(TSommet*)GetItem(Sommet->Utilise,j);
    i=GetIndex(S->UtilisePar,Sommet);
    Extract(S->UtilisePar,i);
  }

  while (Sommet->UtilisePar->Count > 0)
    Remove((TSommet*)GetItem(Sommet->UtilisePar,0));

  i=GetIndex(Dg,Sommet);
  free(Extract(Dg,i));
}

static TSommet* CreateSommet(ESigne Signe, TPredicat* Pred){
  TSommet* S;
  S = make(TSommet);
  S->Signe = Signe;
  S->Predicat = Pred;
  S->Utilise = CreateListe();
  S->UtilisePar = CreateListe();
  return S;
}

static void Calculer(ESigne Signe, TPredicat* Pred){
  TSommet *S,*AncSommetAppelant;
  TRelation R;
  int i;

  for(i=0;i<Dg->Count;i++){
     S=(TSommet*)GetItem(Dg,i);
     if ((S->Predicat == Pred) && (S->Signe == Signe)){
       if (SommetAppelant!=NULL){
         AddToSet(SommetAppelant->Utilise,S);
         AddToSet(S->UtilisePar,SommetAppelant);}
       return;
       }
  }

  AncSommetAppelant = SommetAppelant;
  while (1){
    S = CreateSommet(Signe,Pred);
    Add(Dg,S);
    SommetAppelant = S;

    R = EvalPredicat(Signe,Pred);
```

```
    if (Signe == POS)
      if (Pred->Pos==R){
        DestroyRelation(R);
        break;
      }
      else{
        DestroyRelation(Pred->Pos);
        Pred->Pos = R;
      }
    else
      if (Pred->Neg==R){
        DestroyRelation(R);
        break;
      }
      else{
        DestroyRelation(Pred->Neg);
        Pred->Neg = R;
      }

    if (S->UtilisePar->Count == 0)  break;

    Remove(S);
  }
  SommetAppelant = AncSommetAppelant;
  if (SommetAppelant != NULL){
    AddToSet(SommetAppelant->Utilise,S);
    AddToSet(S->UtilisePar,SommetAppelant);
  }
}

static void InitPointFixe(void){
  TPredicat* Pred;
  int i;
  for(i=0;i<PredTable->Count;i++){
    Pred = GetPredicat(i);
    Pred->Pos = FALSE;
    Pred->Neg = FALSE;
  }
  Dg = CreateListe();
  SommetAppelant = NULL;
}

TRelation CalculerPointFixe(ESigne Signe, char* NomPredicat){
   int i;
   TPredicat *Pred;

   InitPointFixe();
   i=GetNrPredicat(NomPredicat);

   if (i==-1){
      printf("Erreur de requete : le predicat %s n'est pas defini\n",
              NomPredicat);
      exit(1);
   }
   Pred = GetPredicat(i);

   Calculer(Signe,Pred);

   if (Signe==POS)
      return Pred->Pos;
   else
      return Pred->Neg;
}
```

A.7 principal.c

Ce module est le module principal, il exporte la fonction **main** qui est le point d'entrée du programme et un fichier **yyin** qui sera lu par le parseur.

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <stdlib.h>
#include "tables.h"

#include "trelation.h"
#define SOURCE_FILE 0
#define PREDICATE 1
#define TABLE_SIZE 2

extern char* actual_line;
```

yyparse du module **parse.y**

```
int yyparse(void);
```

NumeroterPredicats et NumeroterVars du module **numerotage.c**

```
void NumeroterPredicats(void);
void NumeroterVars(void);
```

Typage du module **typage.c**

```
void Typage(void);
```

CalculerPointFixe du module **pointfixe.c**

```
TRelation CalculerPointFixe(ESigne S, char *Requete);
```

PrintRelation du module **printrelation.c**

```
void PrintRelation(TRelation R);

FILE* yyin;
```

Codage

```
static void ErreurInvocation(void){
  printf("usage : beta -f nom_fichier -p ['+'|'-'|'']nom_predicat [-t table_size]\n");
  exit(1);
}

char* examination(int n, char* t[], int opt) {
  int i;

  for (i=1 ; i<n-1 ; i++) {
    switch (opt) {
    case SOURCE_FILE : if (strcmp(t[i],"-f")==0) {return t[i+1];} ; break;
    case PREDICATE   : if (strcmp(t[i],"-p")==0) {return t[i+1];} ; break;
    case TABLE_SIZE  : if (strcmp(t[i],"-t")==0) {return t[i+1];} ; break;
    }
  }
  if (opt!=TABLE_SIZE) ErreurInvocation();
  return "";
}

int main(int argc,char* argv[])
{
  TRelation R;
  char *File_Name=examination(argc,argv,SOURCE_FILE);
  char *Requete=examination(argc,argv,PREDICATE);
  int MaxNoeuds;

  MaxNoeuds=atoi(examination(argc,argv,TABLE_SIZE));
  if (MaxNoeuds==0) MaxNoeuds=20000;
```

```
  actual_line=(char*) malloc(sizeof(char)*1000);
  InitTables();

  InitRelationTable(MaxNoeuds);

  yyin = fopen(File_Name,"rb");
  if (!yyin){
    printf("erreur d'ouverture fichier : %s\n",File_Name);
    return 1;}
  yyparse();
  fclose(yyin);

  NumeroterPredicats();
  NumeroterVars();
  Typage();

  if (Requete[0] == '+')
     R = CalculerPointFixe(POS,Requete+1);
  else
    if (Requete[0] == '-')
       R = CalculerPointFixe(NEG,Requete+1);
    else
       R = CalculerPointFixe(POS,Requete);

  PrintRelation(R);

  PrintRelationTable(False);

  return 0;
}
```

A.8 printrelation.c

Ce module exporte une seule fonction qui affiche une relation sur *stdout*.

```
#include <stdio.h>
#include <malloc.h>
#define make(T) (T*)malloc(sizeof(T))

#include "tables.h"
#include "trelation.h"
```

Codage

```
static TListe* AddVarsRelation(TRelation R,TListe* VarListe){
  int V, i;

  if (R == TRUE || R == FALSE)
    return VarListe;

  V = GetLabel(R);
  AddToSortedIntSet(VarListe, V);
  for (i=0; i < GetCardVar(V); i++)
    VarListe = AddVarsRelation(GetFils(R)[i],VarListe);

  return VarListe;
}

static TListe* GetVarsRelation(TRelation R){
  return AddVarsRelation(R,CreateListe());
}

static void PrintSuccesBranch
            (TListe* Relations, TListe* Valeurs, TListe* VarListe){
  int Col;
  TRelation R;
  int Val;
  int Variable,NVar;
  char *NomVar;
  int i;
  char* s;

  printf("(");
  Col = 0;

  for (i=0;i<Relations->Count;i++){
    R = *(TRelation*)GetItem(Relations,i);
    Val =  GetInt(Valeurs,i);
    Variable = GetLabel(R);

    while (GetInt(VarListe,Col)<Variable){
      NVar = GetInt(VarListe,Col);
      NomVar = GetString(NomVarTable,NVar);
      if (Col != 0) printf(", ");
      printf("%s:%s",NomVar,"*");
      Col++;
    }

    s = StringOfConst(*GetNDomVar(Variable),Val);
    NomVar=(char*)GetItem(NomVarTable,Variable);
    if (Col != 0) printf(", ");
    printf("%s:%s",NomVar,s);
    free(s);
    Col++;
  }
```

```
  while (Col<=VarListe->Count-1){
    if (Col != 0) printf(", ");
    NVar = GetInt(VarListe,Col);
    NomVar = GetString(NomVarTable,NVar);
    printf("%s:%s",NomVar,"*");
    Col++;
  }
  printf(")\n");
}
void PrintRelation(TRelation R){
  TListe *VarListe;
  TListe *Peres;
  TRelation Pere, Fils, *q;
  TListe *Options;
  int *Option;
  int *p;

  VarListe = GetVarsRelation(R);

  Peres = CreateListe();
  Options=CreateListe();

  if (R==TRUE){
    printf("TRUE\n");
    return;
  }
  if (R==FALSE){
    printf("FALSE\n");
    return;
  }

  q=make(TRelation);*q=R;Add(Peres,q);
  AddToIntListe(Options,0);

  while (Peres->Count != 0){
    Pere = *(TRelation*)GetLast(Peres);
    Option = (int*)GetLast(Options);

    if (*Option >= GetCardVar(GetLabel(Pere))){
      free(ExtractLast(Peres));
      free(ExtractLast(Options));
      if (Peres->Count > 0){
        Option = (int*) GetLast(Options);
        *Option = *Option +1;
      }
      continue;
    }

    Fils = GetFils(Pere)[*Option];
    if (Fils == TRUE){
      PrintSuccesBranch(Peres,Options,VarListe);
      *Option = *Option + 1;
      continue;
    }
    if (Fils == FALSE){
     *Option = *Option + 1;
      continue;
    }
    q = make(TRelation); *q = Fils; Add(Peres, q);
    p = make(int);  *p = 0;  Add(Options,p);
  }
}
```

A.9 tables.h

```
#ifndef TABLES_H
#define TABLES_H

#include "types.h"

void InitTables(void);
```

PredTable : SEQ[TPredicat]

```
extern TListe* PredTable;
int GetNrPredicat(char* Nom);
TPredicat* GetPredicat(int i);
```

DomTable : SEQ[TDomaine]

```
extern TListe* DomTable;
TDomaine* GetDomaine(int i);
char* StringOfConst(int Dom,int NConst);

extern int NVariables;

extern TListe* NomVarTable;

extern int* DomVarTable;
int* GetNDomVar(int V);
TDomaine* GetDomVar(int V);

extern int* CardVarTable;

int GetCardVar(int i);

#endif
```

A.10 tables.c

Ce module exporte les tableaux dans lesquels sont stockées les informations syntaxiques sur le programme à interpréter, ainsi que quelques fonctions pour y accéder plus facilement.

```
#include <malloc.h>
#include <stdio.h>
#include <assert.h>
#include "types.h"
#include<string.h>

TListe* PredTable;
TListe* DomTable;
int NVariables;
TListe* NomVarTable;
int* DomVarTable;
int* CardVarTable;
```

Codage

```
void InitTables(void){
  PredTable=CreateListe();
  DomTable=CreateListe();
  NVariables=0;
  NomVarTable=CreateListe();
  DomVarTable=NULL;
  CardVarTable=NULL;
}

int GetNrPredicat(char*  Nom) {
  int i;
  TPredicat* Pred;

  for (i=0;i<PredTable->Count;i++){
    Pred = (TPredicat*)GetItem(PredTable,i);
    if (strcmp(Pred->Nom,Nom)==0) return i;
  }
  return -1;
}

TDomaine* GetDomaine(int i){
  return (TDomaine*)GetItem(DomTable,i);
}

TPredicat* GetPredicat(int i){
  return (TPredicat*)GetItem(PredTable,i);
}

int* GetNDomVar(int V){
  assert(V<=NVariables);
  return DomVarTable+V;
}

TDomaine* GetDomVar(int V){
  assert(V<=NVariables);
  return GetDomaine(*GetNDomVar(V));
}

int GetCardVar(int V){
  assert(V<=NVariables);
  return CardVarTable[V];
}

char* StringOfConst(int NDom,int NConst){
  TDescrDomaine *DescrDom = GetDomaine(NDom)->Definition;
  TDomIntervalle *DescrIntervalle;
  char *s=(char*)malloc(1000);
```

```
  if (DescrDom->T == Enumere)
    strcpy(s,(char*)GetItem(DescrDom->Def.DescrEnumere,NConst));
  else{
    DescrIntervalle = DescrDom->Def.DescrIntervalle;
    sprintf(s,"%s[%d]",DescrIntervalle->Nom,
                       DescrIntervalle->BInf+NConst);
  }
  return s;
}
```

Regarde si le nom recu en INPUT est un nom de Domaine existant

```
int Nom_of_Domain_Exist(char* Nom) {
  TDomaine* Dom;
  int i;

  for(i=0 ; i<DomTable->Count ; i++) {
    Dom=(TDomaine*) GetItem(DomTable,i);
    if (strcmp(Dom->Nom,Nom)==0) return i;
  }
  return -1;
}

int Nom_of_Constante_Exist_OutThere(char* Name) {
  TDomaine *Dom;
  int i;

  for (i=0 ; i<DomTable->Count ; i++) {
    Dom=(TDomaine*) GetItem(DomTable,i);
    if (Dom->Definition->T==Enumere) {
      if (GetIndexString(Dom->Definition->Def.DescrEnumere,Name)>=0) {
        return i; }
    }
    if (Dom->Definition->T==Intervalle) {
      if (strcmp(Name,Dom->Definition->Def.DescrIntervalle->Nom)==0) {
        return i;}
    }
  }
  return -1;
}

TDomaine* Get_Domaine_of_Intervalle(char* Name, int binf, int bsup) {
  TDomaine* Dom;
  int i;

  for (i=0 ; i<DomTable->Count ; i++) {
    Dom=(TDomaine*) GetItem(DomTable,i);
    if (Dom->Definition->T==Intervalle) {
      if ((strcmp(Dom->Definition->Def.DescrIntervalle->Nom,Name)==0)
          && (binf >= (Dom->Definition->Def.DescrIntervalle->BInf))
          && (bsup <= (Dom->Definition->Def.DescrIntervalle->BSup)))
        return Dom;
    }
  }
  return NULL;
}

int Get_Domaine_of_Constante_Nommee(char* Name) {
  TDomaine* Dom;
  int i;
```

```
  for (i=0 ; i<DomTable->Count ; i++) {
    Dom=(TDomaine*) GetItem(DomTable,i);
    if (Dom->Definition->T==Enumere) {
      if (GetIndexString(Dom->Definition->Def.DescrEnumere,Name)>=0) return i;
    }
  }
  return -1;
}

int Declared_Constante(TConst* C) {
  TDomaine* Dom;
  int i;

  for (i=0 ; i<DomTable->Count ; i++) {
    Dom=(TDomaine*) GetItem(DomTable,i);
    if ((C->T==Elementaire) && (Dom->Definition->T==Enumere)) {
      if (GetIndexString(Dom->Definition->Def.DescrEnumere,C->Def.ConstElem)>=0)
        return 1;
    }
    if ((C->T==Indexee) && (Dom->Definition->T==Intervalle)) {
      if ((C->Def.ConstIndexee->Index >= Dom->Definition->Def.DescrIntervalle->BInf)
          && (C->Def.ConstIndexee->Index <= Dom->Definition->Def.DescrIntervalle->BSup))
        return 1;
    }
  }
  return 0;
}

int Nom_of_Intervalle_Exist(char* Name) {
  TDomaine* Dom;
  int i;

  for(i=0 ; i<DomTable->Count ; i++) {
    Dom=(TDomaine*) GetItem(DomTable,i);
    if (Dom->Definition->T==Intervalle) {
      if (strcmp(Dom->Definition->Def.DescrIntervalle->Nom,Name)==0)
        return i;
    }
  }
  return -1;
}

TDomaine* Get_Domaine_of_Constante(TConst* C) {
  TDomaine* D;

  Reset(DomTable);
  while (!EOL(DomTable)) {
    D=(TDomaine*) GetNext(DomTable);
    if ((C->T==Elementaire) && (D->Definition->T==Enumere))
      if (GetIndexString(D->Definition->Def.DescrEnumere,C->Def.ConstElem)==1) return D;
    if ((C->T==Indexee) && (D->Definition->T==Intervalle)
        && (strcmp(C->Def.ConstIndexee->Nom,D->Definition->Def.DescrIntervalle->Nom)==0)) return D;
  }
  return NULL;
}

int Numero_de_Domaine(char* Name) {
  TDomaine* D;
  int i=0;
```

```
  Reset(DomTable);
  while (!EOL(DomTable)) {
    D=GetNext(DomTable);
    if (strcmp(D->Nom,Name)==0) {
      return i;}
    i++;
  }
  return -1;
}

int Nom_of_Param_Exist(char* Name, TListe *L) {
  TParamType *D;
  int i;

  for (i=0 ; i<L->Count ; i++) {
    D=GetItem(L,i);
    if (strcmp(D->NomVar,Name)==0) {
      return i;}
  }
  return -1;
}
```

A.11 tliste.h

```
#ifndef TLISTE_H
#define TLISTE_H

typedef enum {False, True} Boolean;

typedef struct{
  int Capacity;
  int Count;
  int Current;
  void** List;
} TListe;

TListe* CreateListe(void);
void DestroyListe(TListe* L);

int Count(TListe* L);

void* GetItem(TListe* L,int Index);

void* GetLast(TListe* L);
void* Extract(TListe* L, int Index);
void* ExtractLast(TListe* L);
int GetIndex(TListe* L, void* p);

void Add(TListe* L,void* info);
void AddToSet(TListe* L,void* info);
void Insert(TListe* L,void* info,int i);
void Swap(TListe* L,int i, int j);

void Reset(TListe* L);
void* GetNext(TListe* L);
Boolean EOL(TListe* L);
```

les listes d'entiers

```
int GetInt(TListe* L,int i);
int ExtractInt(TListe* L,int i);
int GetIndexInt(TListe* IntTable, int i);

void AddToIntListe(TListe* L,int i);
void AddToSortedIntSet(TListe* IntSet, int i);
TListe* AppendIntListe(TListe* A,TListe* B);
TListe* CopyIntListe(TListe* L);

void SortIntListe(TListe* L);

char* GetString(TListe* L, int i);
int GetIndexString(TListe* StringTable, char* Nom);

void AddCopyToStringListe(TListe* L, char* s);
void AddToStringSet(TListe* StringSet,char* Nom);
void AddCopyToStringSet(TListe* L, char *s);

TListe* AppendStringListe(TListe* A,TListe* B);
TListe* DiffStringList(TListe* A,TListe* B);

#endif
```

A.12 tliste.c

Ce module implémente les listes.

```
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#define make(t) (t*)malloc(sizeof(t))

typedef enum {False, True} Boolean;

typedef struct{
  int Capacity;
  int Count;
  int Current;
  void** List;
} TListe;
```

Codage

Les listes génériques

```
inline TListe* CreateListe(void){
  TListe* L;
  L = make(TListe);
  L->Count = 0;
  L->Capacity = 0;
  L->List = NULL;
  return L;
}

inline void DestroyListe(TListe* L){
  int i;
  for (i=0;i<L->Count;i++)
    free(L->List[i]);
  free(L->List);
  free(L);
}

inline int Count(TListe* L){
  return L->Count;
}

inline void* GetItem(TListe* L, int Index){
  assert( Index>=0 && Index<L->Count );
  return L->List[Index];
}

inline void* GetLast(TListe* L){
  assert(L->Count>0);
  return L->List[L->Count-1];
}

inline void* Extract(TListe* L, int Index){
  void* p;
  int i;
  assert( Index>=0 && Index<L->Count );
  p = L->List[Index];
  for (i=Index;i<=L->Count-2;i++)
    L->List[i] = L->List[i+1];
  L->Count--;
  return p;
}

inline void* ExtractLast(TListe* L){
  return Extract(L,L->Count-1);
}
```

```
inline int GetIndex(TListe* L, void* p){
  int i;
  for (i=0;i<L->Count;i++)
    if (L->List[i] == p)  return i;
  return -1;
}

void Add(TListe* L, void* info){
  L->Count++;
  if (L->Count > L->Capacity){
    if (L->Capacity <= 4)
      L->Capacity +=4;
    else if (L->Capacity <=8)
      L->Capacity += 8;
    else
      L->Capacity += 16;
    L->List = (void**)
      realloc(L->List,L->Capacity * sizeof(void*));
  }
  L->List[L->Count - 1] = info;
}

inline void AddToSet(TListe* L, void* info){
  if (GetIndex(L,info) < 0)
    Add(L,info);
}

void Insert(TListe* L,void* info,int index){
  int i;
  assert(index>=0 && index<=L->Count);
  L->Count++;
  if (L->Count > L->Capacity){
    if (L->Capacity <= 4)
      L->Capacity +=4;
    else if (L->Capacity <=8)
      L->Capacity += 8;
    else
      L->Capacity += 16;
    L->List = (void**)
        realloc(L->List,L->Capacity * sizeof(void*));
  }

  for (i=L->Count-2;i>=index;i--)
    L->List[i + 1] = L->List[i];

  L->List[index] = info;
}

void Swap(TListe* L,int i, int j){
  void *p;
  assert((i>=0)&&(i<L->Count)&&(j>=0)&&(j<L->Count));
  p = L->List[i];
  L->List[i] = L->List[j];
  L->List[j] = L->List[i];
}

inline void Reset(TListe* L){
  L->Current = 0;
}

inline Boolean EOL(TListe* L){
  if (L->Current >= L->Count)
    return True;
  else return False;
}
```

```
inline void* GetNext(TListe* L){
  assert(! EOL(L));
  return L->List[L->Current++];
}
```

Les listes d'entiers

```
inline int GetInt(TListe* L,int i){
  return *(int*)GetItem(L,i);
}
```

```
inline int ExtractInt(TListe* L,int i){
  int *p,j;
  p = (int*) Extract(L,i);
  j = *p;
  free(p);
  return j;
}
```

```
inline int GetIndexInt(TListe* IntTable, int i){
  int index;
  for (index=0; index < IntTable->Count; index++)
    if (*(int*)GetItem(IntTable,index) == i)
      return index;
  return -1;
}
```

```
inline void AddToIntListe(TListe* L,int i){
  int* p = make(int);
  *p = i;
  Add(L,p);
}
```

```
void AddToSortedIntSet(TListe* IntSet, int i){
  int *p,index,j;
  for (index=0; index < IntSet->Count; index++){
    j = *(int*)GetItem(IntSet,index);
    if (j == i) return;
    if (j > i){
      p = make(int);
      *p = i;
      Insert(IntSet,p,index);
      return;
    }
  }
  p = make(int);
  *p = i;
  Add(IntSet,p);
}
```

```
TListe* AppendIntListe(TListe* A,TListe* B){
   int i;
   TListe* C;
   int* p;
   C = CreateListe();
   for (i=0;i<A->Count;i++){
      p=make(int);
      *p=*(int*)GetItem(A,i);
      Add(C,p);
   }
   for (i=0;i<B->Count;i++){
      p=make(int);
      *p=*(int*)GetItem(B,i);
      Add(C,p);
   }
   return C;
}

TListe* CopyIntListe(TListe* L){
  TListe* L2 = CreateListe();
  int *p,i;

  L2->Capacity = L->Capacity;
  L2->Count = L->Count;
  L2->Current = L->Current;

  L2->List = malloc(L->Capacity * sizeof(void*));
  for (i=0;i<L->Count;i++){
    p = make(int);
    *p =*(int*)(L->List[i]);
    L2->List[i] = p;
  }
  return L2;
}

void SortIntListe(TListe* L){
  int length, i, *max, *suiv;

  length = L->Count;
  for (length = L->Count; length>1;length--){
    i=0;
    max = (int*)(L->List[i]);
    while (i<length-1){
      suiv = (int*)(L->List[i+1]);
      if (*max > *suiv){
        L->List[i]=suiv;
        L->List[i+1]=max;}
      else
        max = suiv;
      i++;
    }
  }
}
```

Les listes de chaînes de charactères

```
inline char* GetString(TListe* L,int i){
  return (char*)GetItem(L,i);
}
```

```
inline int GetIndexString(TListe* StringTable, char* Nom){
  int i;
  char* s;
  for (i = 0;i<StringTable->Count;i++){
    s = (char*) GetItem(StringTable,i);
    if (strcmp(s,Nom)==0) return i;
  }
   return -1;
}

inline void AddCopyToStringListe(TListe* L, char* s){
  char *sc;
  sc = (char*)malloc(strlen(s)+1);
  strcpy(sc,s);
  Add(L,sc);
}

inline void AddToStringSet(TListe* StringSet,char* Nom){
   if (GetIndexString(StringSet,Nom)<0)
     Add(StringSet,Nom);
}

inline void AddCopyToStringSet(TListe* StringSet, char *Nom){
   if (GetIndexString(StringSet,Nom)<0)
     AddCopyToStringListe(StringSet,Nom);
}

TListe* AppendStringListe(TListe* A,TListe* B){
   int i;
   TListe* C;
   char *s,*sc;
   C = CreateListe();
   for (i=0;i<A->Count;i++){
     s=(char*)GetItem(A,i);
     sc=(char*)malloc((strlen(s)+1)*sizeof(char));
     strcpy(sc,s);
     Add(C,sc);
   }
   for (i=0;i<B->Count;i++){
     s=(char*)GetItem(B,i);
     sc=(char*)malloc((strlen(s)+1)*sizeof(char));
     strcpy(sc,s);
     Add(C,sc);
   }
   return C;
}

TListe* DiffStringList(TListe* A,TListe*B){
  int i;
  char *s,*sc;
  TListe* Diff;
  Diff = CreateListe();
  for (i=0;i<A->Count;i++){
    s = (char*)GetItem(A,i);
    if (GetIndexString(B,s)<0){
      sc=(char*)malloc(strlen(s)+1);
      strcpy(sc,s);
      Add(Diff,sc);
    }
  }
  return Diff;
}
```

A.13 trelation.h

```
#ifndef TRELATION_H
#define TRELATION_H

typedef struct{
  int Label;
  int Mult;
  int* Fils;
} TNoeud;

typedef int TRelation;

#define FALSE (TRelation)0
#define TRUE  (TRelation)1

extern TNoeud** RelationTable;

#define GetLabel(R) (RelationTable[R]->Label)
#define GetFils(R) (RelationTable[R]->Fils)

void InitRelationTable(int max);

TRelation CreateRelation(int v, TRelation *fils);

#include <assert.h>
extern TNoeud *EMPTY, *REMOVED;
extern int MAX_NOEUDS;
inline extern TRelation CopyRelation(TRelation i){
  if (i==TRUE||i==FALSE) return i;
  assert(i<MAX_NOEUDS);
  assert(RelationTable[i]!=REMOVED);
  assert(RelationTable[i]!=EMPTY);
  RelationTable[i]->Mult++;
  return i;
}

void DestroyRelation(TRelation i);

void PrintRelationTable(int print);

#endif
```

A.14 trelation.c

Ce module définit le type abstrait **TRelation**

```
#include <assert.h>
#include <stdio.h>
#include <malloc.h>
#include <math.h>
#define make(T) (T*)malloc(sizeof(T))
typedef enum {False, True} Boolean;

int GetCardVar(int v);

typedef struct{
  int Label;
  int Mult;
  int* Fils;
} TNoeud;

#define FALSE 0
#define TRUE  1

TNoeud *EMPTY, *REMOVED;

int MAX_NOEUDS;

static int RelationTable_Count;
static int max_RelationTable_Count;
TNoeud** RelationTable;
```

Codage

```
int FH_9(int v, int n, int* fils){
  unsigned int adr;
  int i;

  adr=0;
  for(i = 0; i < n; i++) adr += (fils[i]<<((i+v)));
  adr= (adr % (MAX_NOEUDS-3))+2;
  return adr;
}

int FH_8(int v, int n, int* fils){
  unsigned int adr;
  int i;

  adr=0;
  for(i = 0; i < n; i++) adr += (fils[i]*fils[i]);
  adr= (adr % (MAX_NOEUDS-3))+2;
  return adr;
}

int FH_7(int v, int n, int* fils){
  unsigned int adr;
  int i;

  adr=0;
  for(i = 0; i < n; i++) adr *= (int) log10((double) (fils[i]));
  adr += v;
  adr= (adr % (MAX_NOEUDS-3))+2;
  return adr;
}

int FH_6(int v, int n, int* fils){
  unsigned int adr;
  int i;
```

```
  adr=0;
  for(i = 0; i < n; i++) adr += (unsigned int) sqrt((float) fils[i]);
  adr += v;
  adr= (adr % (MAX_NOEUDS-3))+2;
  return adr;
}

int FH_5(int v, int n, int* fils){
  unsigned int adr;
  int i;

  adr=0;
  for(i = 0; i < n; i++) adr += (MAX_NOEUDS - fils[i]);
  adr += v;
  adr= (adr % (MAX_NOEUDS-3))+2;
  return adr;
}

int FH_4(int v, int n, int* fils){
  unsigned int adr;
  int i;

  adr=0;
  for(i = 0; i < n; i++) adr += (int) log10((double) (fils[i]<<i));
  adr += v;
  adr= (adr % (MAX_NOEUDS-3))+2;
  return adr;
}

int FH_3(int v, int n, int* fils){
  unsigned int adr;
  int i;

  adr=0;
  for(i = 0; i < n; i++) adr += (fils[i]*fils[i]);
  adr += v;
  adr= (adr % (MAX_NOEUDS-3))+2;
  return adr;
}

int FH_2(int v, int n, int* fils){
  unsigned int adr;
  int i;

  adr=1;
  for(i = 0; i < n; i++) adr *= fils[i];
  adr += v;
  adr= (adr % (MAX_NOEUDS-3))+2;
  return adr;
}

int FH_1(int v, int n, int* fils){
  unsigned int adr;
  int i;

  adr=0;
  for(i = 0; i < n; i++) adr += fils[i];
  adr += v;
  adr= (adr % (MAX_NOEUDS-3))+2;
  return adr;
}

int FH(int v, int n, int* fils){
  unsigned int adr, d;
  int i;
```

```
  adr = v*1973;
  for(i = 0; i < n; i++) adr += fils[i]<<i;
  d = adr << 8;
  adr += d;
  d = d << 8;
  adr += d;
  d = d << 8;
  adr += d;
  adr *= 13;
  adr /= 97;
  adr %= (MAX_NOEUDS - 3);
  adr += 2;
  return adr;
}

static int ComputeBaseAdress(int v, int n, int* fils){
  return FH(v,n,fils);
}

inline int CopyRelation(int i){
  if (i==TRUE||i==FALSE) return i;
  assert(i<MAX_NOEUDS);
  assert(RelationTable[i]!=REMOVED);
  assert(RelationTable[i]!=EMPTY);
  RelationTable[i]->Mult++;
  return i;
}

void DestroyRelation(int i){
  int j;
  TNoeud* noeud;

  if (i==TRUE||i==FALSE) return;
  assert(i<MAX_NOEUDS);
  assert(RelationTable[i]!=REMOVED);
  assert(RelationTable[i]!=EMPTY);
  RelationTable[i]->Mult--;
  if (RelationTable[i]->Mult==0){
    noeud = RelationTable[i];
    for (j=0;j<GetCardVar(noeud->Label);j++)
       DestroyRelation(noeud->Fils[j]);
    free(RelationTable[i]);
    RelationTable[i]=REMOVED;
    RelationTable_Count--;
  }
}

void InitRelationTable(int max){
  int i;

  EMPTY = make(TNoeud);
  REMOVED = make(TNoeud);
  MAX_NOEUDS = max;

  RelationTable = (TNoeud**)malloc(MAX_NOEUDS*sizeof(TNoeud*));
  for (i=0;i<MAX_NOEUDS;i++) RelationTable[i] = EMPTY;
  if (!RelationTable){
    printf("Pas assez de memoire pour la creation de la table des noeuds\n");
    exit(1);
  }
  RelationTable_Count = 0;
  max_RelationTable_Count = 0;
}

inline static void InsertRelation(int v, int* fils, int adr){
  TNoeud* Noeud;
```

```
  assert(adr<MAX_NOEUDS);
  assert(RelationTable[adr]==EMPTY||RelationTable[adr]==REMOVED);

  Noeud = (TNoeud*) malloc(sizeof(TNoeud));
  Noeud->Label = v;
  Noeud->Fils = fils;
  RelationTable[adr] = Noeud;
  RelationTable[adr]->Mult = 1;
  if (RelationTable_Count == max_RelationTable_Count)
    max_RelationTable_Count++;
  RelationTable_Count++;
}

inline static Boolean TestRelation(int v, int n, int* fils, int adr){
  int i;
  TNoeud* Noeud;

  assert(adr<MAX_NOEUDS);
  assert((RelationTable[adr]!=EMPTY)&&(RelationTable[adr]!=REMOVED));
  Noeud = RelationTable[adr];
  if (v != Noeud->Label) return FALSE;
  for (i=0;i<n;i++)
    if (Noeud->Fils[i] != fils[i]) return FALSE;
  return TRUE;
}

int CreateRelation(int v, int* fils){
  Boolean egaux;
  int first, next, i, n, base, libre, adr;

  n=GetCardVar(v);
  egaux = TRUE;
  first = fils[0];
  for (i=1; i<n; i++){
    next = fils[i];
    if (next != first){
      egaux = FALSE;
      break;
    }
  }
  if (egaux){
    if (first!=TRUE && first!=FALSE)
      RelationTable[first]->Mult++;
    free(fils);
    return first;
  }

  base = ComputeBaseAdress(v,n,fils);

  libre = -1;

  for(adr=base; adr<MAX_NOEUDS;adr++){
    if (RelationTable[adr] == EMPTY){
      InsertRelation(v,fils,(libre==-1)?adr:libre);
      return (libre==-1)?adr:libre;
    }
    if (RelationTable[adr] == REMOVED){
      if (libre == -1) libre = adr;
      continue;
    }
    if (TestRelation(v,n,fils,adr)){
      RelationTable[adr]->Mult++;
      free(fils);
      return adr;
    }
  }
```

```
  for(adr=2; adr<base;adr++){
    if (RelationTable[adr] == EMPTY){
      InsertRelation(v,fils,(libre==-1)?adr:libre);
      return (libre==-1?adr:libre);
    }
    if (RelationTable[adr] == REMOVED){
      if (libre == -1) libre = adr;
      continue;
    }
    if (TestRelation(v,n,fils,adr)){
      RelationTable[adr]->Mult++;
      free(fils);
      return adr;
    }
  }

  if (libre == -1){
    printf("La table des noeuds est pleine\n");
    exit(1);
  }
  else{
    InsertRelation(v,fils,libre);
    return libre;
  }
}

void PrintRelationTable(int print){
  int i,j;
  TNoeud *noeud;
  int v;
  int base, wrong, total, maxi, d;
  int partage;

  wrong=0;
  total=0;
  maxi=0;
  partage=0;

  for (i=0;i<MAX_NOEUDS;i++)
    if ((RelationTable[i]!=EMPTY)&&(RelationTable[i]!=REMOVED)){
      partage += RelationTable[i]->Mult;
      noeud = RelationTable[i];
      v = noeud->Label;
      base = ComputeBaseAdress(v,GetCardVar(v),noeud->Fils);
      if (base!=i){
        if (print) printf("*");
        wrong++;
        d=(i>base)?i-base:i+MAX_NOEUDS-base;
        total+=d;
        if (d>maxi) maxi=d;
      }
      else if (print!=0) printf(" ");
      if (print!=0){
        printf("%6d :: (%6d) :: %3d :: #%3d#",i,base,RelationTable[i]->Mult,v);
        for (j=0;j<GetCardVar(noeud->Label);j++)
          printf("%6d ",noeud->Fils[j]);
        printf("\n");
      }
    }
```

```
  printf("\n\n");
  printf("%20s = %10d\n","MAX_NOEUDS",MAX_NOEUDS);
  printf("%20s = %10d\n","RelationTable_Count",RelationTable_Count);
  printf("%20s = %10d\n","max_RelationTable_Count",max_RelationTable_Count);
  printf("%20s = %13.2f\n","facteur de partage",
         ((double)partage)/RelationTable_Count);
  printf("%20s = %10d\n","mal place",wrong);
  printf("%20s = %13.2f\n","distance moyenne",
          ((double)total)/RelationTable_Count);
  printf("%20s = %10d\n","distance maximale",maxi);
}
```

A.15 types.h

Ce fichier définit les types syntaxiques du β-langage (cfr. le chapitre 12 du rapport)

```
#ifndef TYPES_H
#define TYPES_H

#include "tliste.h"
#include "trelation.h"

typedef struct {
  char* Nom;
  int BInf;
  int BSup;
} TDomIntervalle;

typedef enum {Enumere,Intervalle} ETypeDescrDom;

typedef struct{
  ETypeDescrDom T;
  union {
    TListe* DescrEnumere; /* SEQ[String], donc des 'char*'*/
    TDomIntervalle* DescrIntervalle;
  }Def;
} TDescrDomaine;

typedef struct {
  char* Nom;
  TDescrDomaine* Definition;
  int Count;
} TDomaine;

typedef struct {
  char* Nom;
  int Index;
} TConstIndexee;

typedef enum {Elementaire,Indexee} ETypeConst;

typedef struct {
  int Const;
  int Dom;
  ETypeConst T;
  union {
    char* ConstElem;
    TConstIndexee* ConstIndexee;
  } Def;
} TConst;

typedef struct{
  char* NomVar;
  int Var;
} TVar;

typedef enum {Var,Const} ETypeVarOuConst;

typedef struct {
  ETypeVarOuConst T;
  union {
    TVar* Var;
    TConst* Const;} Def;
} TVarOuConst;

typedef struct {
  char* NomPred;
  int Pred;
  int NLigne;
  TListe* ParamsEffs;  /* SEQ[TVarOuConst] */
} TAppel;
```

```
typedef enum {POS,NEG} ESigne;

typedef struct {
  ESigne Signe;
  TAppel* Appel;
  Boolean FirstTime;
  TListe *VarsLit;
  TListe *VarsLitSorted;
  TListe *VarsDef;
  TListe *VarsDefSorted;
  TRelation RelContr;
} TLiteral;

typedef struct {
  char* NomVar;
  int Var;
  int Domaine;
  TListe* Constantes;  /* SEQ[Integer] */
  int NLigne;
} TAppartenance;

typedef enum {EQ=0,NE=1} EOperateur;

typedef struct {
  char* NomVar1;
  int Var1;
  EOperateur Op;
  char* NomVar2;
  int Var2;
  int NLigne;
} TComparVar;

typedef enum {ComparVar, Appartenance} ETypeContrainte;

typedef struct{
   ETypeContrainte T;
   union{
     TComparVar* Comparaison;
     TAppartenance* Appartenance;
   } Def;
} TContrainte;

typedef struct {
  TListe* Contraintes;     /* SEQ[TContrainte] */
  TListe* Literaux;        /* SEQ[TLiteral] */
  TListe* VarsQuants;      /* SEQ[Integer] */
} TConjonction;

typedef struct {
  char* NomVar;
  char* NomDomaine;
  int Domaine;
} TParamType;

typedef struct {
  char* Nom;
  TListe* ParamTypes;      /* SEQ[TParamTypes] */
  TListe* Conjonctions;    /* SEQ[TConjonction]*/
  TListe* Vars;            /* SEQ[Integer] */
  TListe* VarsSorted;            /* SEQ[Integer] */
  TRelation Pos;
  TRelation Neg;
} TPredicat;

#endif
```

A.16 typage.c

Ce module exporte une seule fonction qui permet de déterminer les types des variables du programme source.

```
#include <stdio.h>
#include <stdlib.h>

#include "tables.h"
```

Codage

```
static char* ConstToStr(TConst* C){
  TConstIndexee *Ci;
  char* s;
  if (C->T == Elementaire)
    return C->Def.ConstElem;
  else{
    Ci = C->Def.ConstIndexee;
    s = (char*)malloc(strlen(Ci->Nom)+10+2+1);
    sprintf(s,"%s[%d]",Ci->Nom,Ci->Index);
  }
  return s;
}

static void TypageAppel(TAppel* Appel){
  TPredicat *PredAppele;
  int i,DomAppele,*Dom;
  TVarOuConst *ParamEff;
  TConst *C;
  TVar *V;
  char* s;

  PredAppele = GetPredicat(Appel->Pred);
  for (i=0;i<Appel->ParamsEffs->Count;i++){
    ParamEff = (TVarOuConst*)GetItem(Appel->ParamsEffs,i);
    DomAppele = ((TParamType*)GetItem(PredAppele->ParamTypes,i))->Domaine;

    if (ParamEff->T == Const){
      C = ParamEff->Def.Const;
      if (C->Dom != DomAppele){
        s = ConstToStr(C);
        printf("%s %d %s %s %s %s %s %s\n",
               "Erreur : ligne",Appel->NLigne,": La constante",s,
               "est de type",GetDomaine(C->Dom)->Nom,
               "mais est utilisee comme parametre de type",
               GetDomaine(DomAppele)->Nom);
        free(s);
        exit(1);
      }
    }
    else{  /* ParamEff-¿T == Var */
      V = ParamEff->Def.Var;
      Dom = GetNDomVar(V->Var);
      if ((*Dom != -1) && (*Dom != DomAppele)){
        printf("%s %d %s %s %s %s %s %s %s\n",
               "Erreur : Ligne",Appel->NLigne,
               ": Conflit de type pour la variable",V->NomVar,":",
               GetDomaine(*Dom)->Nom,"ou",GetDomaine(DomAppele)->Nom,"?");
        exit(1);
      }
      *Dom=DomAppele;
    }
  }
}
```

```
static void TypageAppartenance(TAppartenance* Appart){
  int *NDomVar;

  NDomVar = GetNDomVar(Appart->Var);

  if ((*NDomVar != -1) && (*NDomVar != Appart->Domaine)){
    printf("%s %d %s %s %s %s %s %s %s\n",
           "Erreur : Ligne", Appart->NLigne,
           ": Conflit de type pour la variable",Appart->NomVar, ":",
           GetDomaine(*NDomVar)->Nom,"ou",GetDomaine(Appart->Domaine)->Nom,
           "?");
    exit(1);
  }
  *NDomVar = Appart->Domaine;
}

static Boolean TypageCompVar(TComparVar* Comp){
  int Var1,Var2,*NDomVar1,*NDomVar2;

  Var1 = Comp->Var1;
  Var2 = Comp->Var2;
  NDomVar1 = GetNDomVar(Var1);
  NDomVar2 = GetNDomVar(Var2);

  if ((*NDomVar1 != -1) && (*NDomVar2 != -1) && (*NDomVar1 != *NDomVar2)){
    printf("%s %d %s %s %s %s %s %s?\n",
           "Erreur : Ligne", Comp->NLigne,
           ": Conflit de type pour les variables",
           Comp->NomVar1, Comp->NomVar2,
           GetDomaine(*NDomVar1)->Nom,"ou",GetDomaine(*NDomVar2)->Nom);
    exit(1);
  }

  if ((*NDomVar1 == -1) && (*NDomVar2 != -1)){
    *NDomVar1 = *NDomVar2;
    return True;
  }

  if ((*NDomVar1 != -1) && (*NDomVar2 == -1)){
    *NDomVar2 = *NDomVar1;
    return True;
  }

  return False;
}

void TypageConjonction(TConjonction* Conj){
  TLiteral* Lit;
  Boolean Modif;
  TContrainte* Contr;
  TComparVar* Comp;
  int *NDomVar1,*NDomVar2;
  int i;

  for (i=0;i<Conj->Literaux->Count;i++){
    Lit=(TLiteral*)GetItem(Conj->Literaux,i);
    TypageAppel(Lit->Appel);
  }

  for (i=0;i<Conj->Contraintes->Count;i++){
    Contr=(TContrainte*)GetItem(Conj->Contraintes,i);
    if (Contr->T != Appartenance) continue;
    TypageAppartenance(Contr->Def.Appartenance);
  }
```

```
  do{
    Modif = False;
    for (i=0;i<Conj->Contraintes->Count;i++){
      Contr=(TContrainte*)GetItem(Conj->Contraintes,i);
      if (Contr->T != ComparVar) continue;
      Modif = Modif || TypageCompVar(Contr->Def.Comparaison);
    }
  } while (Modif);

  for (i=0;i<Conj->Contraintes->Count;i++){
    Contr=(TContrainte*)GetItem(Conj->Contraintes,i);
    if (Contr->T != ComparVar) continue;
    Comp = Contr->Def.Comparaison;
    NDomVar1 = GetNDomVar(Comp->Var1);
    NDomVar2 = GetNDomVar(Comp->Var2);
    if ((*NDomVar1 == -1) && (*NDomVar2 == -1)){
      printf("%s %d %s %s %s %s %s\n",
             "Erreur : Ligne",Comp->NLigne, ": Le type des variables",
             Comp->NomVar1,"et", Comp->NomVar2,"est indeterminable");
      exit(1);
    }
  }
}

void TypagePredicat(TPredicat* Pred){
  int i,NVar,*NDomVar;
  TParamType* ParamType;
  TConjonction* Conj;

  for (i=0;i<Pred->Vars->Count;i++){
    NVar = *(int*)GetItem(Pred->Vars,i);
    NDomVar = GetNDomVar(NVar);
    ParamType = (TParamType*)GetItem(Pred->ParamTypes,i);
    *NDomVar = ParamType->Domaine;
  }

  for (i=0;i<Pred->Conjonctions->Count;i++){
    Conj=(TConjonction*)GetItem(Pred->Conjonctions,i);
    TypageConjonction(Conj);
  }
}

void Typage(void){
  TPredicat* Pred;
  int i;
  DomVarTable=(int*)malloc(NVariables*sizeof(int));
  for (i=0;i<NVariables;i++)
    DomVarTable[i]=-1;
  for (i=0;i<PredTable->Count;i++){
    Pred=(TPredicat*)GetItem(PredTable,i);
    TypagePredicat(Pred);
  }
  CardVarTable=(int*)malloc(NVariables*sizeof(int));
  for (i=0;i<NVariables;i++)
    CardVarTable[i]=GetDomVar(i)->Count;
}
```

B Le code CaML pour l'interprétation abstraite

B.1 Les prédicats auxilliaires

```
let soi = string_of_int;;

let rec M i j =
        if j = 0
   then "M"^(string_of_int i)
   else (M i (j-1)) ^ "'";;

let Mpar p = fun i -> M i p;;

let rec comb f i j s=
   if i = j
   then (f i)
   else if i < j
        then (f i)^s^(comb f (i+1) j s)
        else "erreur";;

let params f i j = comb f i j ",";;

let Mparams i j p = params (Mpar p) i j;;



let domdef = "domain {g,f,v,a};\n\n";;

let dtrans = "domain dtrans  {v_to_a, v_to_vg, void};\n\n";;

let trans_v =
        "trans_v(M,M',T:dtrans) := \n" ^
   "\t M <> v & M' = M |\n" ^
   "\t M = v & T = void & M' = v |\n" ^
   "\t M = v & T = v_to_a & M' = a |\n" ^
   "\t M = v & T = v_to_vg & M' in {v,g};\n\n";;

let trans_vf_to_a =
   "trans_vf_to_a(M,M') := \n" ^
   "\t M in {v,f} & M' = a |\n" ^
   "\t M in {g,a} & M' = M;\n\n";;


let ttrans_1 =
        "ttrans_1(M,M',T:dtrans) := \n" ^
   "\t M in {g,f} & T = void |\n" ^
   "\t M in {v,a} & M' in {f,v} & T = void |\n" ^
   "\t M in {v,a} & M' = g & T = v_to_vg |\n" ^
   "\t M in {v,a} & M' = a & T = v_to_a;\n\n";;

let ttrans_k k =
        "ttrans_" ^ (soi k) ^ "(" ^ (Mparams 1 k 0) ^ "," ^ (Mparams 1 k 1) ^
          ",T:dtrans) := \n" ^
        "\t ttrans_1(M1,M1',T) & T in {v_to_vg,v_to_a} |\n" ^
        "\t ttrans_1(M1,M1',void) & ttrans_" ^ (soi (k-1)) ^
                                "(" ^ (Mparams 2 k 0) ^ "," ^
              (Mparams 2 k 1) ^ ",T);\n\n";;
```

```
let ttrans k =
        if k = 1
   then ttrans_1
   else ttrans_k k;;


let unif_var_aux =
"unif_var_aux(M1,M2,M) := \n" ^
"\t M1 = g & M = g | \n" ^
"\t M2 = g & M = g | \n" ^
"\t M1 <> g & M2 <> g & M1 = a & M = a | \n" ^
"\t M1 <> g & M2 <> g & M2 = a & M = a | \n" ^
"\t M1 in {f,v} & M2 in {f,v} & M = v; \n\n";;


let unif_var_n n =
"unif_var_" ^
(soi n) ^
"(" ^
(Mparams 1 n 0) ^
"," ^
(Mparams 1 n 1) ^
") := \n" ^
"\t unif_var_aux(M1,M2,M) &\n" ^
"\t M1' = M & M2' = M & ttrans_2(M1,M2,M1',M2',T) & \n" ^
"\t " ^
(comb (fun i->
              "trans_v(" ^ (M i 0) ^ "," ^ (M i 1) ^ ",T)"
      )
      3 n " & "
) ^
";\n\n";;


let unif_var_2  =
"unif_var_2(M1,M2,M1',M2') := \n" ^
"\t unif_var_aux(M1,M2,M) &\n" ^
"\t M1' = M & M2' = M;  \n\n";;


let unif_var n =
        if n = 2
   then unif_var_2
   else unif_var_n n;;


let testg k =
"testg_" ^ (soi k) ^ "(" ^
(Mparams 1 k 0) ^ ") :=\n\t" ^
(comb  (fun i ->
                (M i 0) ^ " = g")
       1 k " & ") ^
";\n\n";;
```

```
let testva k =
"testva_" ^ (soi k) ^ "(" ^
(Mparams 1 k 0) ^ ") := \n\t" ^
(comb  (fun i ->
                (M i 0) ^ " in {v,a}")
       1 k " & ") ^
";\n\n";;

let dtrans_col3 = "domain dtrans_col3  {all_to_g3, vf_to_a3, void3};\n\n";;

let col3 =
"col3(M,T:dtrans_col3) := \n" ^
"\t M = g & T = all_to_g3 |\n" ^
"\t M in {f,v} & T = void3 |\n" ^
"\t M = a & T = vf_to_a3; \n\n";;


let trans_col3 =
"trans_col3(M,M',T:dtrans_col3) := \n" ^
"\t T = all_to_g3 & M' = g |\n" ^
"\t T = vf_to_a3 & M in {v,f} & M' = a | \n" ^
"\t T = vf_to_a3 & M in {g,a} & M' = M | \n" ^
"\t T = void3 & M' = M; \n\n";;


let dtrans_col4 = "domain dtrans_col4  {v_to_a4, v_to_gv4, vf_to_a4, void4};\n\n";;

let col4_cas2 =
"col4_cas2(M,T:dtrans_col4) := \n" ^
"\t M in {g,f} & T = void4 |\n" ^
"\t M in {v,a} & T = v_to_a4;\n\n";;


let col4_cas3 =
"col4_cas3(M,T:dtrans_col4) := \n" ^
"\t M = g & T = v_to_gv4 |\n" ^
"\t M = f & T = void4 |\n" ^
"\t M = f & T = v_to_a4 |\n" ^
"\t M = a & T = vf_to_a4;\n\n";;


let trans_col4 =
"trans_col4(M,M',T:dtrans_col4) := \n" ^
"\t T = v_to_a4 & M = v & M' = a |\n" ^
"\t T = v_to_a4 & M <> v & M' = M |\n" ^
"\t T = v_to_gv4 & M = v & M' in {g,v} |\n" ^
"\t T = v_to_gv4 & M <> v & M' = M |\n" ^
"\t T = vf_to_a4 & M in {v,f} & M' = a |\n" ^
"\t T = vf_to_a4 & M in {g,a} & M' = M |\n" ^
"\t T = void4 & M' = M ;\n\n" ;;


let unif_func_1_q q =
"unif_func_1_" ^(soi q) ^ "(" ^
(Mparams 1 (1+q) 0) ^ "," ^ (Mparams 1 (1+q) 1) ^ ") := \n" ^
```

```
"\t\t M1'=g & \n" ^
"\t\t ttrans_1(M1,M1',T) &\n" ^
"\t\t " ^ (comb (fun i ->
                  "trans_v(" ^ (M i 0) ^ "," ^ (M i 1) ^ ",T)")
               2 (1+q) " & "
         ) ^ " ;\n\n";;


let unif_func_1_0 =
"unif_func_1_0(M1,M1') := M1'=g\n\n";;


let unif_func_p_0 p =
"unif_func_" ^ (soi p) ^ "_0" ^ "(" ^
(Mparams 1 p 0) ^ "," ^ (Mparams 1 p 1) ^ ") := \n" ^

"\t testg_" ^ (soi (p-1)) ^ "(" ^ (Mparams 2 p 0) ^ ") & \n" ^
"\t\t " ^ (comb (fun i-> (M i 1) ^ "=g")
               1 p " & ") ^ " | \n" ^

"\t not testg_" ^ (soi (p-1)) ^ "(" ^ (Mparams 2 p 0) ^
    ") &\n" ^
"\t\t col3(M1,T3) & \n" ^
"\t\t " ^
   (comb (fun i -> "trans_col3(" ^ (M i 0) ^ "," ^ (M i 1) ^ ",T3)" )
         2 p " & " ) ^ " &\n " ^
"\t\t trans_vf_to_a(M1,M1') ; \n\n";;



let unif_func_p_q p q =
"unif_func_" ^ (soi p) ^ "_" ^ (soi q) ^ "(" ^
(Mparams 1 (p+q) 0) ^ "," ^ (Mparams 1 (p+q) 1) ^ ") := \n" ^

"\t testg_" ^ (soi (p-1)) ^ "(" ^ (Mparams 2 p 0) ^ ") & \n" ^
"\t\t " ^ (comb (fun i-> (M i 1) ^ "=g")
               1 p " & ") ^ " & \n" ^
"\t\t ttrans_1(M1,M1',T) &\n" ^
"\t\t " ^ (comb (fun i ->
                  "trans_v(" ^ (M i 0) ^ "," ^ (M i 1) ^ ",T)")
               (p+1) (p+q) " & "
         ) ^ " |\n" ^

"\t not testg_" ^ (soi (p-1)) ^ "(" ^ (Mparams 2 p 0) ^
    ") & not testva_" ^ (soi (p-1)) ^ "(" ^ (Mparams 2 p 0) ^ ") &\n" ^
"\t\t col3(M1,T3) & col4_cas2(M1,T4) & \n" ^
"\t\t " ^
   (comb (fun i -> "trans_col3(" ^ (M i 0) ^ "," ^ (M i 1) ^ ",T3)" )
         2 p " & " ) ^ " &\n " ^
"\t\t " ^
   (comb (fun i -> "trans_col4(" ^ (M i 0) ^ "," ^ (M i 1) ^ ",T4)" )
         (p+1) (p+q) " & " ) ^ " &\n " ^
"\t\t trans_vf_to_a(M1,M1') | \n" ^
```

```
"\t testva_" ^ (soi (p-1)) ^ "(" ^ (Mparams 2 p 0) ^ ") &\n" ^
"\t\t col3(M1,T3) & col4_cas3(M1,T4) & \n" ^
"\t\t " ^
   (comb (fun i -> "trans_col3(" ^ (M i 0) ^ "," ^ (M i 1) ^ ",T3)" )
         2 p " & " ) ^ " &\n " ^
"\t\t " ^
   (comb (fun i -> "trans_col4(" ^ (M i 0) ^ "," ^ (M i 1) ^ ",T4)" )
         (p+1) (p+q) " & " ) ^ " &\n " ^
"\t\t trans_vf_to_a(M1,M1');\n\n" ;;


let unif_func p q =
        if p = 1
   then if q = 0
        then unif_func_1_0
        else unif_func_1_q q
   else if q = 0
        then unif_func_p_0 p
        else unif_func_p_q p q;;
```

B.2 La traduction

```
include "pred_aux.ml";;

exception toto;;

let max i j = if i > j then i else j;;

let rec maxl = fun
      []      -> raise toto
    | [x]     -> x
    | (x::xs) -> max x (maxl xs);;


let rec length = fun
  [] -> 0
| (x::xs) -> 1+(length xs);;

let rec mapping f = fun
   [] -> []
 | (x::xs) -> (f x)::(mapping f xs);;

let rec interval i j = if i=j then [i] else i::(interval (i+1) j);;

let rec enlever x = fun
   [] -> []
 | (y::ys) -> if x=y then ys else (y::(enlever x ys));;

let rec diff xs = fun
   [] -> xs
 | (y::ys) -> diff (enlever y xs) ys;;

let rec add x = fun
   [] -> [x]
 | (y::ys) -> y::(if x=y then ys else (add x ys));;

let clean_set = fun
     [] -> []
   | (x::xs) -> add x xs;;

let rec elem x =  fun
   [] -> false
 | (y::ys) -> if x=y then true else elem x ys;;

let rec get i = fun
   [] -> raise toto
 | (x::xs) -> if (i=1) then x else (get (i-1) xs);;

let rec union xs = fun
      [] -> xs
    | (y::ys) -> add y (union xs ys);;

let merge (s, res) (s2,res2) = (s^s2, union res res2);;
```

```
type atome = UVAR of int*int
           | UFUNC of int*string*(int list)
           | APPEL of string*(int list);;

type goal == atome list;;

type clause == string*int*goal;;

type proc == string*int*(goal list);;

type prog == proc list;;



type res = Unif_var of int
         | Unif_func of int * int
         | Extg of int * int;;


let rec concc = fun
   [] -> ""
 | [s] -> s
 | (s::ss) -> s^","^(concc ss);;


let parsl l n = concc (mapping (Mpar n) l);;

let trad_uvar nom_atome n i j=
   let l = i::j::(diff (interval 1 n) [i;j]) in
   let s =
      nom_atome ^
      "(" ^
      (Mparams 1 n 0) ^
      "," ^
      (Mparams 1 n 1) ^
      ") := \n\t" ^
      "unif_var_" ^
      (soi n) ^
      "(" ^
      (parsl l 0) ^
      "," ^
      (parsl l 1)^
      ");\n" in
   (s,[Unif_var n]);;


let trad_ufunc nom_atome n is=
   let js = diff (interval 1 n) is in
   let p = length is in
   let q = length js in
   let l = is @ js in
   let s =
      nom_atome ^
```

```
      "(" ^
      (Mparams 1 n 0) ^
      "," ^
      (Mparams 1 n 1) ^
      ") := \n\t" ^
      "unif_func_" ^
      (soi p) ^
      "_" ^
      (soi q) ^
      "(" ^
      (parsl 1 0) ^
      "," ^
      (parsl 1 1)^
      ");\n" in
  (s,[Unif_func(p,q)])
      ;;


let trad_appel nom_atome n p is =
   let js = diff (interval 1 n) is in
   let ijs = is @ js in
   let k = length is in
   let l = length js in
   let s =
      nom_atome ^
      "(" ^
      (Mparams 1 n 0) ^
      "," ^
      (Mparams 1 n 1) ^
      ") := \n\t" ^
      p ^
      "(" ^
      (parsl is 0) ^
      "," ^
      (parsl is 1) ^
      ") & \n\t" ^
      "ttrans_" ^
      (soi k) ^
      "(" ^
      (parsl is 0) ^
      "," ^
      (parsl is 1) ^
      ",T) & \n\t" ^
      (comb
            (fun r->
                     let x = get r js in
                     "trans_v(" ^ (M x 0) ^ "," ^ (M x 1) ^ ",T)"
                     )
            1 l " & "
            ) ^
      ";\n"  in
  (s,[Extg (k,l)])
      ;;


let trad_atome nom_atome n = fun
```

```
   (UVAR (i,j)) -> trad_uvar nom_atome n i j
 | (UFUNC (i,f,is)) -> trad_ufunc nom_atome n (i::is)
 | (APPEL (p,is)) ->  trad_appel nom_atome n p is;;



let max i j = if i > j then i else j;;

let rec maxl = fun
   [x] -> x
 | (x::xs) -> max x (maxl xs);;

let rec var_max_clause = fun
   [] -> 0
 | ((UVAR (i,j))::al) -> max (max i j) (var_max_clause al)
 | ((UFUNC (i,f,is))::al) -> max (maxl (i::is)) (var_max_clause al)
 | ((APPEL (p,is))::al) -> max (maxl is) (var_max_clause al);;


let rec extc k n =
    if n = k
    then ""
    else (Mpar 0 (k+1)) ^
         "=f & " ^
         (extc (k+1) n);;

let rec trad_clause_aux nom_clause n i r =
    "\t" ^
    nom_clause ^
    "_" ^
    (soi i) ^
    "(" ^
    (Mparams 1 n (i-1)) ^
    "," ^
    (Mparams 1 n i) ^
    ")" ^
    (if i = r then ";"
              else " & \n" ^
                   (trad_clause_aux nom_clause n (i+1) r));;




let rec trad_list_atomes nom_clause n i = fun
    [] -> ("",[])
  | (a::al) -> merge (trad_atome (nom_clause ^ "_" ^ (soi i)) n a)
             (trad_list_atomes nom_clause n (i+1) al);;



let trad_clause nom_clause k al =
    let n = var_max_clause al in
    let r = length al in
    let s =
    nom_clause ^
```

```
    "(" ^
    (Mparams 1 k 0) ^
    "," ^
    (Mparams 1 k r) ^
    ") := " ^
    (extc k n) ^
    "\n" ^
    (trad_clause_aux nom_clause n 1 r) ^
    "\n" in
    merge (s,[]) (trad_list_atomes nom_clause n 1 al) ;;


let rec trad_proc_aux nom_proc n i r =
    "\t" ^
    nom_proc ^
    "_" ^
    (soi i) ^
    "(" ^
    (Mparams 1 n 0) ^
    "," ^
    (Mparams 1 n 1) ^
    ")" ^
    (if i = r then ";"
              else " | \n" ^
                   (trad_proc_aux nom_proc n (i+1) r));;

let rec trad_list_clauses nom_proc n i = fun
    [] -> ("",[])
  | (c::cs) ->merge (merge (trad_clause (nom_proc ^ "_" ^(soi i)) n c) ("\n",[]))
            (trad_list_clauses nom_proc n (i+1) cs);;


let trad_proc nom_proc n goals =
    let k = length goals in
    let s =
    nom_proc ^
    "(" ^
    (Mparams 1 n 0) ^
    "," ^
    (Mparams 1 n 1) ^
    ") := \n" ^
    (trad_proc_aux nom_proc n 1 k) ^
    "\n\n" in
    merge (s,[]) (trad_list_clauses nom_proc n 1 goals);;

let rec trad_prog_aux = fun
   [] -> ("\n",[])
 | ((p,n,gs)::prs) -> merge (merge (trad_proc p n gs) ("\n",[]))
                            (trad_prog_aux prs);;


let rec get_unif_vars = fun
    [] -> []
  | (x::xs) -> match x with
                 (Unif_var n) -> n::(get_unif_vars xs)
               | _            -> get_unif_vars xs;;
```

```
let rec get_unif_funcs = fun
    [] -> []
  | (x::xs) -> match x with
                 (Unif_func n) -> n::(get_unif_funcs xs)
               | _             -> get_unif_funcs xs;;


let rec get_extgs = fun
    [] -> []
  | (x::xs) -> match x with
                 (Extg n) -> n::(get_extgs xs)
               | _          -> get_extgs xs;;


let rec print_unif_var = fun
    [] -> ""
  | (n::ns) -> (unif_var n) ^ (print_unif_var ns);;


let rec print_unif_func = fun
    [] -> ""
  | ((p,q)::rs) -> (unif_func p q) ^ (print_unif_func rs);;

let rec print_test = fun
    [] -> ""
  | (p::ps) ->
          if p > 1
          then (testg (p-1)) ^ (testva (p-1)) ^ (print_test ps)
          else print_test ps;;

let rec print_ttrans k =
   if k = 0
   then ""
   else (ttrans k) ^ (print_ttrans (k-1));;


let print_auxs res =
   let unif_funcs = get_unif_funcs res in
   let unif_vars  = get_unif_vars  res in
   let extgs      = get_extgs      res in

   (print_unif_var unif_vars) ^

   (print_unif_func unif_funcs) ^

   (print_test  (clean_set (mapping fst unif_funcs))) ^

   (print_ttrans (max (maxl (mapping fst extgs)) 2));;



let trad_prog prog =
   let (s,res) = trad_prog_aux prog in
```

```
   domdef ^
   dtrans ^
   dtrans_col3 ^
   dtrans_col4 ^

   s ^

   "//predicats auxilliares :\n\n" ^

   trans_v ^
   trans_vf_to_a ^
   unif_var_aux ^
   col3 ^
   trans_col3 ^
   col4_cas2 ^
   col4_cas3 ^
   trans_col4 ^

   (print_auxs res)
;;



let prog_test =
   [("natPlus",3,
             [
             [UFUNC (1,"O",[]); UVAR (2,3)];
             [UFUNC (1,"s",[4]); UFUNC(3,"s",[5]);
                                  APPEL("natPlus",[4;2;5])]
             ]);
    ("natMul",3,
             [
             [UFUNC(1,"O",[]); UFUNC(3,"O",[])];
             [UFUNC(1,"s",[4]); APPEL("natMul",[4;2;5]);
                                APPEL("natPlus",[2;5;3])]
             ])
   ];;



print_string (trad_prog prog_test);;
```